```
/*MDL_BEGIN
! MDL_SOURCE_FILE_NAME: RICKS_R96MATH.H
! PKG_ACRONYM          : R96MA! MDL_SOURCE_VERSION# : V1.0
! MDL_DESCRIPTION:
!  This module is an include file that provides some of the
!  math support for TTI's REAL96 math datatype.
!***************************************************************************
!*  Copyright (c) 2001 by                                                  *
!*  Touch Technologies Inc., San Diego, California.                        *
!*  ALL RIGHTS RESERVED.                                                   *
!***************************************************************************
! R96 datatype example implementation
!     Data stored in Little endian
!        bits  0-63 are the Integer Part (IP)    (signed, so 63 bits of precision)
!        bits 64-95 are the Fractional Part (FP) (signed, modulus 100,000,000 - ie. 8 digits)
!
!        Sign is generally stored in both IP and FP, however,
!        determine the sign of a r96, as follows:
!        If IP != 0 then IP is 2s complement and can be used to determine the sign
!        If IP == 0 then FP is 2s complement
!        Because zero has no sign, to determine the sign if IP=0, you have to
!        look at the sign stored in fractional part.
!
/* In specific embodiments, routines are provided for unsigned mutliprecision
```

Integer Math (64, 96, and 128 bit)

```
MACROs and routines are organized herein so that 64, 96, and 128 bit variants are in order for each
routine or MACRO.  This makes maintenance easier because the variants are very similar.
Most math is done in unsigned 96bit integer format (scaled). Conversion to 96bit unsigned integer state is
done by removing the sign and then multiplying the IP by 100,000,000 and adding in the FP. For multiply
and divide, some 128 bit math may be required to prevent bit loss.  Optimizations have been performed to
minimize the number of bits required, especially when multiplication is required.
There are MACROs to perform I96 compares, additions and subtractions.  Most MACROs assume that the
parameters are of type (unsigned long *)
```

Miscellaneous MACROs

```
There are MACROs to perform comparisons such as ==, !=, >, <, and compares against ZERO.  There are also
MACROs to copy an integer and to negate an integer.
```

Bit shift MACROs

```
There are MACROs to perform right and left bit shift with single bit shifts and counted shifts.  Counted
shifts MUST be 31 bits or less.  There are MACROs to check for overflow as well used in multiplication.
```

Addition MACROS:

```
If a 2 operand ADD is needed then the SUM and the second ADDEND must be the same variable and NOT the
first ADDEND, i.e.  a = a + b is coded as add (b, a, a).
There are MACROs to handle overflow as well necessary for multiply.  The i##add32 variant adds a 32 bit
integer to the appropriate extended precision integer providing a shortcut for that case
```

Subtraction MACROs:

```
!        These MACROs perform unsigned subtraction.
```

Multiplication Routines:

```
!        The basic mechanism is to determine whether the multiplicand or the
!        multiplier is smaller and use that number as the multiplier.  The
!        multiplier is then shifted left until bit ZERO is set.  Each time
!        the multiplier is shifted left, the multiplicand is shifted right.
!        When bit ZERO is set in the multiplier, the current shifted
!        multiplicand is added to the product.  When the multiplier becomes
!        ZERO, the multiplication stops and the product is now complete.
!
!        We do the multiplication by BIG "switch" statement that does 8
!        bits at a time to save time over doing a single bit at a time.
!        The "ixxmultiply" and "ixxmultiply_oiverflow" routines are created by
!        a program named "create_ixxmultiply_r96math.c".  It does the 64, 96,
!        and 128 bit routines.
!
!        There are also routines to test for and handle overflow.  Overflow is
!        identical to non-overflow except the shift and add routines called
!        check for the overflow condition.
!
!        There are also some special MACROS that multiply specifically by 10,
!        10,000, and 100,000,000.  These speed up the various R96 conversions
!        and operations.
!
```

```
!       NOTE: These routines are 3 operand routines!!! One CANNOT use the same variable for the !
product as either the multiplicand or the multiplier!!!
!
!       There is a set of routines to locate the highest set bit.  These are
!       used to optimize overflow handling in multiplication.  It turns out that
!       overflow can be easily predicted by adding the highest set bits and
!       comparing them against the number of the sign bit.  >= will be overflow.
!       The final result is also compared to see if the sign bit is set which
!       implies overflow before sign handling is done.
!
```

! Integer Divides

```
!       The algorithm shifts the divisor over until it is >= the dividend and
!       then shifts it back one bit.  The divisor is then subtracted from the
!       dividend.  The quotient gets the value of the number of bit shifts that
!       were done added to it.  The difference of the subtract becomes the
!       remainder.  This process is repeated until the remainder is less than
!       the divisor.
!
```

R96 miscellaneous routines and MACROs

```
!    r96abs is a MACRO that converts an R96 number into a positive value
!    r96negate is a MACRO that negates an R96 number
!    r96copy is a MACRO that copies an R96 number from one location to another
!    r96cvtr96toi96 is a routine that converts an R96 number into an unsigned
!    96 bit integer.  The sign is returned as the function value (0 if positive
!    and 1 if negative).  The IP is multiplied by 100,000,000 and then the
!    FP is added in.
!    r96cvtr96toi96_4digitfp is a special case routine of r96cvtr96toi96 that
!    converts an R96 number into an unsigned 96 bit integer when modulus 10,000
!    of the FP == ZERO.  The sign is returned as the function value (0 if
!    positive and 1 if negative).  The IP is multiplied by 10,000 and then the
!    FP is added in.
!    r96cvti96tor96 is a routine that converts an unsigned 96 bit integer
!    into an R96 number.  The sign is a parameter.  The I96 value is
!    divided by 100,000,000.  The quotient becomes the IP and the remainder is
!    the FP.
!    r96cvtasctobin converts an ASCIZ string into an R96 number.
!    r96cvtbintoasc converts an R96 number into an ASCIZ string.
!       If this routine is to be used as part of the TTI R96 math stuff,
!       it should have an optimization that uses a table lookup where the
!       the binary is divided by 10,000 converting 4 digits at a time,
!       instead of 10 which only converts 1 digit at a time.
```

! R96 division

```
!    Divide is accomplished by converting the dividend and divisor to 96bit
!    integers and then doing a 96bit divide.  The remainder is then converted
!    to a 128 bit integer and multiplied by 100,000,000.  This value is then
!    divided by the original divisor using the least required bits to get the
!    FP portion of the result quotient.  Overflow is tested by checking bit
!    63 of the result quotient.  Overflow can happen when dividing by a
!    fractional entity, i.e. 100 / .1 == 100 * 10.
!
!    We call r96divide_nohilong as an optimization when the IP part of both
!    numbers is < 2^32.  This routine is able to save 32 bits of precision
!    in the operations.
!
!    Overflow handling is done by checking the unsigned final quotient
!    (IP portion) against 0x7FFFFFFFFFFFFFFF before signs are applied.
!    If the value is larger, then runtime$error is called to signal the
!    error.  The runtime$error in the R96 math .H files is intended
!    to be over-written.  The definition is conditional.
!
```

R96 multiplication

```
!    The multiplication starts with a conversion to 128 bit unsigned
!    integers.  a 128 bit multiply is then done.  The product is
!    then divided by (100,000,000 * 100,000,000) to get the IP part
!    of the product.  The remainder of that divide is divided by
!    100,000,000 to get the FP part of the product.
!
!    We start by checking for possible optimizations.  If both FPs == ZERO
!    then we call r96multiply_nofp which performs I64 math.  If either
!    the multiplicand or the multiplier == ZERO, then we call
```

```
!    r96multiply_nofp_multiplicand or r96multiply_nofp_multiplier as
!    appropriate.  Finally, if modulus 10,000 of both FPs == 0 then we call
!    r96multiply_4digitfp.  The last 3 optimization routines perform I96 bit
!    math instead of I128 bit thus running faster.
!
!    Overflow handling is done by checking each left shift and addition for
!    overflow as well as checking the unsigned final product (IP portion)
!    against 0x7FFFFFFFFFFFFFFF before signs are applied.  If the value is
!    larger, then runtime$error is called to signal the error and processing
!    continues until a solution is arrived at.  The runtime$error in the R96
!    math .H files is intended to be over-written.  The definition is
!    conditional.  THIS IMPLIES THAT runtime$error CAN BE CALLED MULTIPLE
!    TIMES.  Noticing an overflow does not directly stop the processing of
!    the multiplication.
*/
#include "ixxmath_nomultiply.h"
#include "ixxmath_multiply.h"
#include "r96math.h"
/*MDL_END*/


      *                        C O P Y R I G H T                         *
      * Copyright (C) 2001                                               *
      * by Touch Technologies Inc.,    San Diego, California             *
      * This software is furnished under a license and may be used and   *
      * copied only in accordance with the terms of such license and with *
      * the inclusion of the above copyright notice. This software or any *
      * other copies thereof may not be provided or otherwise made       *
      * available to any other person. No title or ownership of the      *
      * software is hereby transferred                                   *
      *******************************************************************
      FACILITY: SheerPower Language
      ABSTRACT:
          This module provides the real integer 18.8 math class for
          both Alpha & Intel Pentium Processors.*/
```

Module for the real integer 18.8 math class for Alpha & Intel Pentium

```
#include "c-universal.h"
#include <stdio.h>
#include <string.h>
#include "c-intmsg.h"
#include <time.h>
#include <math.h>
#include "ricks_r96math.h"
typedef struct
  {   int wholedigits;        /* number of whole digits eg. 99.678 has 2 whole digits*/
      int fractdigits;          /* number of fractional digits eg. 99.678000 has 3 fractional digits*/
     int neg;               /* if the number is negative*/
     int rlen;              /* result length eg. 99.678 has an rlen of 6*/
  } realinfo;
long ran_random ();
int randomize ();
void scaledown (void *_a, int n);
int ator(real *a, char *b);
void rtoa (real *, void *, int fdigits, realinfo *rinfo, int flags);
inline void scaleup (void *_a);
static real scale = 1;
#define LARGEST_NUMBER (~(((int64)1)<<63))
//////////////////////////////////////////////////////////////////////////////
// MTH$DEXP DOUBLE EXPONENTIAL
//////////////////////////////////////////////////////////////////////////////
// Brief description: Return the exponential of the input value.
// Expected:  d          =   the address of a 'double'
//    Result:  return    =   the exponential of 'd' as a double
//////////////////////////////////////////////////////////////////////////////
double routine mth$dexp (double *d)
{  return exp (*d);        // call exp() passing the value of 'd'}
//////////////////////////////////////////////////////////////////////////////
// MTH$DLOG2 DOUBLE BASE 2 LOGARITHM
//////////////////////////////////////////////////////////////////////////////
// Brief description: Return the Base 2 logarithm of the input value.
// Expected:  d          =   the address of a 'double'
//    Result:  return    =   the base 2 logarithm of 'd' as a double
```

```
////////////////////////////////////////////////////////////////////////
double routine mth$dlog2 (double *a)
{  return log (*a) * 1.442695099;}
////////////////////////////////////////////////////////////////////////
// MTH$DNINT DOUBLE TO INTEGRAL PART
////////////////////////////////////////////////////////////////////////
// Brief description: Remove the fractional part from a double and return result.
// Method: Convert the double to a large 64-bit integral value. This removes
// the fractional part. Then convert the large 64-bit integer back
// to a double, and return that value.
// Expected:  a           =   the address of a 'double'
//   Result:  return      =   the integral part of 'a' as a double.
////////////////////////////////////////////////////////////////////////
double routine mth$dnint (double *a)
{  int64 i = (int64) * a;      // convert to a large interger
   double d = (double) i;      // then to a double
   return d;                      // return integral part as a double}
////////////////////////////////////////////////////////////////////////
// OTS$POWDD RAISE A DOUBLE TO A DOUBLE EXPONENT
////////////////////////////////////////////////////////////////////////
// Brief description: Raise a double 'x' to a double exponent 'y'.
// Expected:  x           =   the base double
//         :  y           =   the exponent double
//   Result:  return      =   double 'x' raised by 'y'
////////////////////////////////////////////////////////////////////////
double routine ots$powdd (double x, double y)
{  double r = pow(x, y);
   return r;}
////////////////////////////////////////////////////////////////////////
// OTS$POWJJ RAISE AN INTEGER TO AN INTEGER EXPONENT
////////////////////////////////////////////////////////////////////////
// Brief description: Raise an integer 'x' to a integer exponent 'y'.
// Expected:  x           =   the base integer
//         :  y           =   the exponent integer
//   Result:  return      =   double 'x' raised by 'y'
////////////////////////////////////////////////////////////////////////
double routine ots$powjj (int _x, int _y)
{  double y = _y;
   double x = _x;           // convert to double's first for pow()
   double r = pow(x, y);
   return r;}
////////////////////////////////////////////////////////////////////////
// OTS$POWDJ RAISE A DOUBLE TO AN INTEGER EXPONENT
////////////////////////////////////////////////////////////////////////
// Brief description: Raise a double 'x' to an integer exponent 'y'.
// Expected:  x           =   the base double
//         :  y           =   the exponent integer
//   Result:  return      =   double 'x' raised by 'y'
////////////////////////////////////////////////////////////////////////
double routine ots$powdj (double x, int _y)
{  double y = _y;           // convert to double first
   double r = pow(x, y);
   return r;}
////////////////////////////////////////////////////////////////////////
// MTH$DSQRT SQUARE ROOT OF A DOUBLE
////////////////////////////////////////////////////////////////////////
// Brief description: The square root of a double.
// Expected:  d           =   the address of the double
//   Result:  return      =   the square root of 'd' as a double
////////////////////////////////////////////////////////////////////////
double routine mth$dsqrt (double *d)
{  return sqrt (*d);}
////////////////////////////////////////////////////////////////////////
// MTH$DLOG NATURAL LOGARITHM
////////////////////////////////////////////////////////////////////////
// Brief description: The natural logarithm of a double as a double.
// Expected:  d           =   the address of the double
//   Result:  return      =   the natural logaithm of 'd' as a double
////////////////////////////////////////////////////////////////////////
double routine mth$dlog (double *d)
{  return log (*d);}
////////////////////////////////////////////////////////////////////////
// MTH$DLOG10 COMMON LOGARITHM
////////////////////////////////////////////////////////////////////////
// Brief description: The common logarithm of a double.
// Expected:  d           =   the address of the double
//   Result:  return      =   the common logarithm of 'd' as a double
////////////////////////////////////////////////////////////////////////
double routine mth$dlog10 (double *d)
{  return log10 (*d);}
```

// MTH$DSIN THE SINE OF ITS RADIAN ARGUMENT

```
// Brief description: return the sine of its radian argument
// Expected:  d          =  the address of the double
//    Result:  return     =  the return the sine of its radian
//                           argument 'd' as a double
double routine mth$dsin (double *d)
{  return sin (*d);        // return the sine of its radian argument}
```

// MTH$DSINH THE HYPERBOLIC SINE OF ITS RADIAN ARGUMENT

```
// Brief description: return the hyperbolic sine of its radian argument
// Expected:  d          =  the address of the double
//    Result:  return     =  return the hyperbolic sine of its radian
//                           argument 'd' as a double
double routine mth$dsinh (double *d)
{  return sinh (*d);       // return the hyperbolic sine of its radian argument}
```

// MTH$DCOS THE COSINE OF ITS RADIAN ARGUMENT

```
// Brief description: return the cosine of its radian argument
// Expected:  d          =  the address of the double
//    Result:  return     =  return the cosine of its radian
//                           argument 'd' as a double
double routine mth$dcos (double *d)
{  return cos (*d);        // return the cosine of the radian argument}
```

// MTH$DCOSH THE HYPERBOLIC COSINE OF ITS RADIAN ARGUMENT

```
// Brief description: return the hyperbolic cosine of its radian argument
// Expected:  d          =  the address of the double
//    Result:  return     =  return the hyperbolic cosine of its radian
//                           argument 'd' as a double
double routine mth$dcosh (double *d)
{  return cosh (*d);       // return the hyperbolic cosine of its radian argument}
```

// MTH$DTAN THE TANGENT OF ITS RADIAN ARGUMENT

```
// Brief description: return the tangent of its radian argument
// Expected:  d          =  the address of the double
//    Result:  return     =  return the tagent of its radian
//                           argument 'd' as a double
double routine mth$dtan (double *d)
{  return tan (*d);        // return the tangent of its radian argument}
```

// MTH$DTANH THE HYPERBOLIC TANGENT OF ITS RADIAN ARGUMENT

```
// Brief description: return the hyperbolic tangent of its radian argument
// Expected:  d          =  the address of the double
//    Result:  return     =  return the hyperbolic tangent of its radian
//                           argument 'd' as a double
double routine mth$dtanh (double *d)
{  return tanh (*d);       // return the hyperbolic tangent of its radian argument}
```

// MTH$DATAN THE ARC TANGENT OF ITS RADIAN ARGUMENT

```
// Brief description: return the arc tangent of its radian argument
// Expected:  d          =  the address of the double
//    Result:  return     =  return the arc tagent of its radian
//                           argument 'd' as a double
double routine mth$datan (double *d)
{  return atan (*d);       // return the arc tangent of its radian argument}
```

// MTH$DASIN THE ARC SINE OF ITS RADIAN ARGUMENT

```
// Brief description: return the arc sine of its radian argument
// Expected:  d          =  the address of the double
//    Result:  return     =  return the arc sine of its radian
//                           argument 'd' as a double
double routine mth$dasin (double *d)
{  return asin (*d);       // return the arc sine of its radian argument }
```

// MTH$DACOS THE ARC COSINE OF ITS RADIAN ARGUMENT

```
////////////////////////////////////////////////////////////////////////////////
// Brief description: return the arc cosine of its radian argument
// Expected:  d          =   the address of the double
//   Result:  return     =   return the arc cosine of its radian
//                           argument 'd' as a double
////////////////////////////////////////////////////////////////////////////////
double routine mth$dacos (double *d)
{  return acos (*d);        // return the arc cosine of its radian argument}
////////////////////////////////////////////////////////////////////////////////
```

// MTH$DABS THE ABSOLUTE VALUE OF A COMPLEX NUMBER

```
////////////////////////////////////////////////////////////////////////////////
// Brief description: return the absolute value of a complex number
// Expected:  d          =   the address of the double
//   Result:  return     =   return the absolute value of a complex
//                           number as a double
////////////////////////////////////////////////////////////////////////////////
double routine mth$dabs (double *d)
{  return fabs (*d);}
////////////////////////////////////////////////////////////////////////////////
```

// LIB$EDIV EXTENDED PRECISION DIVIDE

```
////////////////////////////////////////////////////////////////////////////////
// Brief description: divide a 64-bit quadword by an integer giving an integer quotent
//  and remainder.
// Expected:  a        =   integer divisor
//            b        =   address of 64-bit quadword
//            c        =   address of integer quotent
//            d        =   address of integer remainder
//   Result:  status from lib$ediv
////////////////////////////////////////////////////////////////////////////////
c_routine int routine lib$ediv (void *a, void *b, void *c, void *d)
{  *(int *) c = (int) (*(int64 *) b / *(int *) a);
   *(int *) d = (int) (*(int64 *) b % *(int *) a);
   return 1;}
////////////////////////////////////////////////////////////////////////////////
```

// EMUL EXTENDED PRECISION MULTIPLY

```
////////////////////////////////////////////////////////////////////////////////
// Brief description: multiply two integers giving a quadword result.
// Expected:
//            a        =   integer multiplier
//            b        =   integer multiplicand
//            c        =   integer addend
//            q        =   address of quadword result
//   Result:  status from lib$emul
////////////////////////////////////////////////////////////////////////////////
int routine lib$emul (int *_a, int *_b, int *c, void *_d)
{  int64 a = *_a;
   int64 b = *_b;
   int64 d = a * b;
   if (c)
      d += *c;
   *(int64 *) _d = d;
   return 1;}
////////////////////////////////////////////////////////////////////////////////
```

// EDIV EXTENDED PRECISION DIVIDE

```
////////////////////////////////////////////////////////////////////////////////
// Brief description: divide a 64-bit quadword by an integer giving an integer quotent
//  and remainder.
// Expected:  a        =   integer divisor
//            b        =   address of 64-bit quadword
//            c        =   address of integer quotent
//            d        =   address of integer remainder
//   Result:  status from lib$ediv
////////////////////////////////////////////////////////////////////////////////
c_routine int routine ediv (int a, int *b, int *c, int *d)
{  return lib$ediv (&a, b, c, d);}
////////////////////////////////////////////////////////////////////////////////
```

// EMUL EXTENDED PRECISION MULTIP

```
////////////////////////////////////////////////////////////////////////////////
// Brief description: multiply two integers giving a quadword result.
// Expected:
//            a        =   integer multiplier
//            b        =   integer multiplicand
//            c        =   integer addend
//            q        =   address of quadword result
//   Result:  status from lib$emul
////////////////////////////////////////////////////////////////////////////////
c_routine int routine emul (int a, int b, int c, int *q)
```

```
{  return lib$emul (&a, &b, &c, q);}
```

// SUBX EXTENDED PRECISION SUBTRACT

```
// Brief description: subtract two 64-bit quadword giving quadword result
// Expected:
//            a        =   address of 64-bit quadword
//            b        =   address of 64-bit quadword
//            c        =   address of 64-bit quadword
//   Result:  status from lib$subx
c_routine int routine lib$subx (void *a, void *b, void *c)
{  *(int64 *) c = *(int64 *) a - *(int64 *) b;
   return 1;}
```

// LIB$ADDX EXTENDED PRECISION ADD

```
// Brief description: add two 64-bit quadword giving quadword result
// Expected:
//            a        =   address of 64-bit quadword
//            b        =   address of 64-bit quadword
//            c        =   address of 64-bit quadword
//   Result:  status from lib$addx
c_routine int routine lib$addx (void *a, void *b, void *c, void*)
{  *(int64 *) c = *(int64 *) a + *(int64 *) b;
   return 1;}
```

RANDOM NUMBERS

// MTH$RANDOM GET A RANDOM VALUE

```
// Brief description: Get a random value modulus 'maxvalue' or if no 'maxvalue'
// return a value between 0 and 1.
// Expected:  maxvalue = address of an integer
//   Result:  floating random number
real routine mth$random (int *maxvalue)
{  int r = ran_random ();      /* get random integer*/
   real one = 1;
   real f;
   if (*maxvalue)
     {         r %= *maxvalue;     /* remainder after divide by maxvalue*/
        f = (real) (r + 1);    /* relative to one*/     }
   else
     {         f = (real) r;
        while (f >= one)       /* make the random number less than one*/
           f /= 10;       }
   return f;                 /* return floa/real*/}
#include    <stdio.h>
/* random.c:
```

An improved random number generation package.

/* In addition to the standard rand()/srand() like interface, this package also has a special state info interface. The initstate() routine is called with a seed, an array of bytes, and a count of how many bytes are being passed in; this array is then initialized to contain information for random number generation with that much state information. Good sizes for the amount of state information are 32, 64, 128, and 256 bytes. The state can be switched by calling the setstate() routine with the same array as was initiallized with initstate(). By default, the package runs with 128 bytes of state information and generates far better random numbers than a linear congruential generator. If the amount of state information is less than 32 bytes, a simple linear congruential R.N.G. is used.

Internally, the state information is treated as an array of longs; the zeroeth element of the array is the type of R.N.G. being used (small integer); the remainder of the array is the state information for the R.N.G. Thus, 32 bytes of state information will give 7 longs worth of state information, which will allow a degree seven polynomial. (Note: the zeroeth word of state information also has some other information stored in it -- see setstate() for details).

The random number generation technique is a linear feedback shift register approach, employing trinomials (since there are fewer terms to sum up that way). In this approach, the least significant bit of all the numbers in the state table will act as a linear feedback shift register, and will have period 2^deg - 1 (where deg is the degree of the polynomial being used, assuming that the polynomial is irreducible and primitive). The higher order bits will have longer periods, since their values are also influenced by pseudo-random carries out of the lower bits. The total period of the generator is approximately

```
deg*(2**deg - 1); thus doubling the amount of state information has a vast influence on the period of the
generator.
Note: the deg*(2**deg - 1) is an approximation only good for large deg, when the period of the shift
register is the dominant factor.  With deg equal to seven, the period is actually much longer than the
7*(2**7 - 1) predicted by this formula.
For each of the currently supported random number generators, we have a break value on the amount of state
information (you need at least this many bytes of state info to support this random number generator), a
degree for the polynomial (actually a trinomial) that the R.N.G. is based on, and the separation between
the two lower order coefficients of the trinomial.*/
///////////////////////////////////////////////////////////////////////////////

#define     TYPE_0      0      /* linear congruential*/
#define     BREAK_0     8
#define     DEG_0       0
#define     SEP_0       0
#define     TYPE_1      1      /* x**7 + x**3 + 1*/
#define     BREAK_1     32
#define     DEG_1       7
#define     SEP_1       3
#define     TYPE_2      2      /* x**15 + x + 1*/
#define     BREAK_2     64
#define     DEG_2       15
#define     SEP_2       1
#define     TYPE_3      3      /* x**31 + x**3 + 1*/
#define     BREAK_3     128
#define     DEG_3       31
#define     SEP_3       3
#define     TYPE_4      4      /* x**63 + x + 1*/
#define     BREAK_4     256
#define     DEG_4       63
#define     SEP_4       1
/* * Array versions of the above information to make code run faster -- relies
 * on fact that TYPE_i == i.*/
#define     MAX_TYPES   5      /* max number of types above*/
static int degrees[MAX_TYPES] = { DEG_0, DEG_1, DEG_2,DEG_3, DEG_4};
static int seps[MAX_TYPES] = { SEP_0, SEP_1, SEP_2,SEP_3, SEP_4};
/* * Initially, everything is set up as if from :
 *       initstate( 1, &randtbl, 128 );
 * Note that this initialization takes advantage of the fact that ran_srandom()
 * advances the front and rear pointers 10*rand_deg times, and hence the
 * rear pointer which starts at 0 will also end up at zero; thus the zeroeth
 * element of the state information, which contains info about the current
 * position of the rear pointer is just
 *   MAX_TYPES*(rptr - state) + TYPE_3 == TYPE_3.*/
static long randtbl[DEG_3 + 1] = { TYPE_3,
   0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
   0xde3b81e0, 0xdf0a6fb5, 0xf103bc02, 0x48f340fb,
   0x7449e56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd,
   0x8c2e680f, 0xeb3d799f, 0xb11ee0b7, 0x2d436b86,
   0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
   0xd7158fd6, 0x6fa6f051, 0x616e6b96, 0xac94efdc,
   0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
   0xf5ad9d0e, 0x8999220b, 0x27fb47b9};
/* * fptr and rptr are two pointers into the state info, a front and a rear
 * pointer.  These two pointers are always rand_sep places aparts, as they cycle
 * cyclically through the state information.  (Yes, this does mean we could get
 * away with just one pointer, but the code for random() is more efficient this
 * way).  The pointers are left positioned as they would be from the call
 *          initstate( 1, randtbl, 128 )
 * (The position of the rear pointer, rptr, is really 0 (as explained above
 * in the initialization of randtbl) because the state table pointer is set
 * to point to randtbl[1] (as explained below).*/
static long *fptr = &randtbl[SEP_3 + 1];
static long *rptr = &randtbl[1];
/* * The following things are the pointer to the state information table,
 * the type of the current generator, the degree of the current polynomial
 * being used, and the separation between the two pointers.
 * Note that for efficiency of random(), we remember the first location of
 * the state information, not the zeroeth.  Hence it is valid to access
 * state[-1], which is used to store the type of the R.N.G.
 * Also, we remember the last location, since this is more efficient than
```

```
 * indexing every time to find the address of the last element to see if
 * the front and rear pointers have wrapped.*/
static long *state = &randtbl[1];
static int rand_type = TYPE_3;
static int rand_deg = DEG_3;
static int rand_sep = SEP_3;
static long *end_ptr = &randtbl[DEG_3 + 1];
/* * ran_srandom:
 * Initialize the random number generator based on the given seed.  If the
 * type is the trivial no-state-information type, just remember the seed.
 * Otherwise, initializes state[] based on the given "seed" via a linear
 * congruential generator.  Then, the pointers are set to known locations
 * that are exactly rand_sep places apart.  Lastly, it cycles the state
 * information a given number of times to get rid of any initial dependencies
 * introduced by the L.C.R.N.G.
 * Note that the initialization of randtbl[] for default usage relies on
 * values produced by this routine.*/
void ran_srandom (unsigned x)
{  register int i, j;
   if (rand_type == TYPE_0)
     {        state[0] = x;      }
   else
     {        j = 1;
        state[0] = x;
        for (i = 1; i < rand_deg; i++)
         {             state[i] = 1103515245 * state[i - 1] + 12345;          }
        fptr = &state[rand_sep];
        rptr = &state[0];
        for (i = 0; i < 10 * rand_deg; i++)
           ran_random ();
     }}
/* * initstate:
 * Initialize the state information in the given array of n bytes for
 * future random number generation.  Based on the number of bytes we
 * are given, and the break values for the different R.N.G.'s, we choose
 * the best (largest) one we can and set things up for it.  ran_srandom() is
 * then called to initialize the state information.
 * Note that on return from ran_srandom(), we set state[-1] to be the type
 * multiplexed with the current value of the rear pointer; this is so
 * successive calls to initstate() won't lose this information and will
 * be able to restart with setstate().
 * Note: the first thing we do is save the current state, if any, just like
 * setstate() so that it doesn't matter when initstate is called.
 * Returns a pointer to the old state.*/
char *ran_initstate (unsigned seed, /* seed for R. N. G.*/
                char *arg_state, /* pointer to state array*/
                int n)      /* # bytes of state info*/
{  register char *ostate = (char *) (&state[-1]);
   if (rand_type == TYPE_0)
      state[-1] = rand_type;
   else
      state[-1] = MAX_TYPES * (rptr - state) + rand_type;
   if (n < BREAK_1)
     {        if (n < BREAK_0)
         {             fprintf (stderr,
                    "initstate: not enough state (%d bytes) with which to do jack; ignored.",
                    n);
            return 0;         }
        rand_type = TYPE_0;
        rand_deg = DEG_0;
        rand_sep = SEP_0;      }
   else
     {        if (n < BREAK_2)
         {             rand_type = TYPE_1;
            rand_deg = DEG_1;
            rand_sep = SEP_1;       }
        else
         {             if (n < BREAK_3)
              {             rand_type = TYPE_2;
                 rand_deg = DEG_2;
                 rand_sep = SEP_2;
```

```
            }
          else
            {                 if (n < BREAK_4)
                {                   rand_type = TYPE_3;
                   rand_deg = DEG_3;
                   rand_sep = SEP_3;
                }
               else
                {                   rand_type = TYPE_4;
                   rand_deg = DEG_4;
                   rand_sep = SEP_4;
                }}}}
   state = &(((long *) arg_state)[1]); /* first location*/
   end_ptr = &state[rand_deg];   /* must set end_ptr before ran_srandom*/
   ran_srandom (seed);
   if (rand_type == TYPE_0)
      state[-1] = rand_type;
   else
      state[-1] = MAX_TYPES * (rptr - state) + rand_type;
   return (ostate);}
/* * setstate:
 * Restore the state from the given state array.
 * Note: it is important that we also remember the locations of the pointers
 * in the current state information, and restore the locations of the pointers
 * from the old state information.  This is done by multiplexing the pointer
 * location into the zeroeth word of the state information.
 * Note that due to the order in which things are done, it is OK to call
 * setstate() with the same state as the current state.
 * Returns a pointer to the old state information.*/
char *ran_setstate (char *arg_state)
{  register long *new_state = (long *) arg_state;
   register int type = new_state[0] % MAX_TYPES;
   register int rear = new_state[0] / MAX_TYPES;
   char *ostate = (char *) (&state[-1]);
   if (rand_type == TYPE_0)
      state[-1] = rand_type;
   else
      state[-1] = MAX_TYPES * (rptr - state) + rand_type;
   switch (type)
     {      case TYPE_0:
      case TYPE_1:
      case TYPE_2:
      case TYPE_3:
      case TYPE_4:
         rand_type = type;
         rand_deg = degrees[type];
         rand_sep = seps[type];
         break;
      default:
         fprintf (stderr,
               "setstate: state info has been munged; not changed.");      }
   state = &new_state[1];
   if (rand_type != TYPE_0)
     {        rptr = &state[rear];
        fptr = &state[(rear + rand_sep) % rand_deg];      }
   end_ptr = &state[rand_deg];   /* set end_ptr too*/
   return (ostate);}
/* * random:
 * If we are using the trivial TYPE_0 R.N.G., just do the old linear
 * congruential bit.  Otherwise, we do our fancy trinomial stuff, which is the
 * same in all ther other cases due to all the global variables that have been
 * set up.  The basic operation is to add the number at the rear pointer into
 * the one at the front pointer.  Then both pointers are advanced to the next
 * location cyclically in the table.  The value returned is the sum generated,
 * reduced to 31 bits by throwing away the "least random" low bit.
 * Note: the code takes advantage of the fact that both the front and
 * rear pointers can't wrap on the same call by not testing the rear
 * pointer if the front one has wrapped.
 * Returns a 31-bit random number.*/
long routine ran_random ()
{  long i;
```

```
    if (rand_type == TYPE_0)
      {         i = state[0] = (state[0] * 1103515245 + 12345) & 0x7fffffff;     }
    else
      {         *fptr += *rptr;
         i = (*fptr >> 1) & 0x7fffffff; /* chucking least random bit*/
         if (++fptr >= end_ptr)
          {            fptr = state;
            ++rptr;        }
         else
          {            if (++rptr >= end_ptr)
              rptr = state;
          }     }
    return (i);}
static unsigned char const ascii_to_ebcdic[] =
{  0, 01, 02, 03, 067, 055, 056, 057,
  026, 05, 045, 013, 014, 015, 016, 017,
  020, 021, 022, 023, 074, 075, 062, 046,
  030, 031, 077, 047, 034, 035, 036, 037,
  0100, 0117, 0177, 0173, 0133, 0154, 0120, 0175,
  0115, 0135, 0134, 0116, 0153, 0140, 0113, 0141,
  0360, 0361, 0362, 0363, 0364, 0365, 0366, 0367,
  0370, 0371, 0172, 0136, 0114, 0176, 0156, 0157,
  0174, 0301, 0302, 0303, 0304, 0305, 0306, 0307,
  0310, 0311, 0321, 0322, 0323, 0324, 0325, 0326,
  0327, 0330, 0331, 0342, 0343, 0344, 0345, 0346,
  0347, 0350, 0351, 0112, 0340, 0132, 0137, 0155,
  0171, 0201, 0202, 0203, 0204, 0205, 0206, 0207,
  0210, 0211, 0221, 0222, 0223, 0224, 0225, 0226,
  0227, 0230, 0231, 0242, 0243, 0244, 0245, 0246,
  0247, 0250, 0251, 0300, 0152, 0320, 0241, 07,
  040, 041, 042, 043, 044, 025, 06, 027,
  050, 051, 052, 053, 054, 011, 012, 033,
  060, 061, 032, 063, 064, 065, 066, 010,
  070, 071, 072, 073, 04, 024, 076, 0341,
  0101, 0102, 0103, 0104, 0105, 0106, 0107, 0110,
  0111, 0121, 0122, 0123, 0124, 0125, 0126, 0127,
  0130, 0131, 0142, 0143, 0144, 0145, 0146, 0147,
  0150, 0151, 0160, 0161, 0162, 0163, 0164, 0165,
  0166, 0167, 0170, 0200, 0212, 0213, 0214, 0215,
  0216, 0217, 0220, 0232, 0233, 0234, 0235, 0236,
  0237, 0240, 0252, 0253, 0254, 0255, 0256, 0257,
  0260, 0261, 0262, 0263, 0264, 0265, 0266, 0267,
  0270, 0271, 0272, 0273, 0274, 0275, 0276, 0277,
  0312, 0313, 0314, 0315, 0316, 0317, 0332, 0333,
  0334, 0335, 0336, 0337, 0352, 0353, 0354, 0355,
  0356, 0357, 0372, 0373, 0374, 0375, 0376, 0377};
static unsigned char const ebcdic_to_ascii[] =
{  0, 01, 02, 03, 0234, 011, 0206, 0177,
  0227, 0215, 0216, 013, 014, 015, 016, 017,
  020, 021, 022, 023, 0235, 0205, 010, 0207,
  030, 031, 0222, 0217, 034, 035, 036, 037,
  0200, 0201, 0202, 0203, 0204, 012, 027, 033,
  0210, 0211, 0212, 0213, 0214, 05, 06, 07,
  0220, 0221, 026, 0223, 0224, 0225, 0226, 04,
  0230, 0231, 0232, 0233, 024, 025, 0236, 032,
  040, 0240, 0241, 0242, 0243, 0244, 0245, 0246,
  0247, 0250, 0133, 056, 074, 050, 053, 041,
  046, 0251, 0252, 0253, 0254, 0255, 0256, 0257,
  0260, 0261, 0135, 044, 052, 051, 073, 0136,
  055, 057, 0262, 0263, 0264, 0265, 0266, 0267,
  0270, 0271, 0174, 054, 045, 0137, 076, 077,
  0272, 0273, 0274, 0275, 0276, 0277, 0300, 0301,
  0302, 0140, 072, 043, 0100, 047, 075, 042,
  0303, 0141, 0142, 0143, 0144, 0145, 0146, 0147,
  0150, 0151, 0304, 0305, 0306, 0307, 0310, 0311,
  0312, 0152, 0153, 0154, 0155, 0156, 0157, 0160,
  0161, 0162, 0313, 0314, 0315, 0316, 0317, 0320,
  0321, 0176, 0163, 0164, 0165, 0166, 0167, 0170,
  0171, 0172, 0322, 0323, 0324, 0325, 0326, 0327,
  0330, 0331, 0332, 0333, 0334, 0335, 0336, 0337,
  0340, 0341, 0342, 0343, 0344, 0345, 0346, 0347,
```

```
  0173, 0101, 0102, 0103, 0104, 0105, 0106, 0107,
  0110, 0111, 0350, 0351, 0352, 0353, 0354, 0355,
  0175, 0112, 0113, 0114, 0115, 0116, 0117, 0120,
  0121, 0122, 0356, 0357, 0360, 0361, 0362, 0363,
  0134, 0237, 0123, 0124, 0125, 0126, 0127, 0130,
  0131, 0132, 0364, 0365, 0366, 0367, 0370, 0371,
  060, 061, 062, 063, 064, 065, 066, 067,
  070, 071, 0372, 0373, 0374, 0375, 0376, 0377};
/* * lib$tra_asc_ebc -- translate ASCII to EBCDIC
 **/
int routine lib$tra_asc_ebc(DESC_S *source, DESC_S *dest)
{  unsigned char *asc, *ebc;
   int slen, dlen;
   int i;
   split$desc (source, &slen, &asc);
   split$desc (dest, &dlen, &ebc);
   for (i = 0; i < slen && i < dlen; i++)
      ebc[i] = ascii_to_ebcdic[asc[i]];

   return SS$_NORMAL;}
/* * lib$tra_ebc_asc -- translate EBCDIC to ASCII
 **/
int routine lib$tra_ebc_asc(DESC_S *source, DESC_S *dest)
{  unsigned char *asc, *ebc;
   int slen, dlen;
   int i;
   split$desc (source, &slen, &ebc);
   split$desc (dest, &dlen, &asc);
   for (i = 0; i < slen && i < dlen; i++)
      asc[i] = ebcdic_to_ascii[ebc[i]];
   return SS$_NORMAL;}
/* * Convert packed decimal to leading separate
 *
 * string: pointer to output ascii string buffer
 *  packed: pointer to input packed decimal string*/
int routine cvtps(char *string, char *packed)
{  int leading = 1;
   int negative = 0;
   char buf[32+1];
   char *s = buf;
   int i, c;
   for (i = 0; i < 32; i++) {     if (i&1) /* odd is lower*/
         c = packed[i/2] & 0x0f;
      else      /* even*/
         c = (packed[i/2]>>4) & 0x0f;
      if (leading && c == 0)
         continue;    /* skip leading 0's*/
      leading = 0;    /* nolonger leading*/
      if (c == 10 || c == 12 || c == 14 || c == 15)
         break;       /* plus*/
      else if (c == 11 || c == 13) {         negative = 1;
         break;          }
      else if (c > 9)
         break;
      *s++ = (char)(c + '0');
      }
   *s++ = '\0';
   if (negative)
      *string++ = '-';  /* no unary '+'*/
   for (i = 0; buf[i]; i++)
      *string++ = buf[i];
   if (i == 0)
      *string++ = '0';  /* at least one zero*/
   *string++ = '\0';
   return SUCCESS;}
/* *  Convert leading separate to packed decimal
 **/
int routine cvtsp(char *packed, char *string, int len)
{  int negative = 0;
   char buf[32+1];
   char *s = buf;
```

```
    int i;
    for (i = 0; i < len && s < buf+32; i++) {     if (string[i] <= ' ')
          continue;
       else if (string[i] == '.')
          continue;
       else if (string[i] == '-')
          negative = 1;
       else if (string[i] == '+')
          negative = 0;
       else if (string[i] >= '0' && string[i] <= '9')
          *s++ = string[i];
       }
    *s++ = '\0';
    len = strlen(buf);
    /*     * if first zero padding required*/
    if ((len&1) == 0) {     for (i = 0; buf[i]; i++);
       for (; i >= 0; --i)
          buf[i+1] = buf[i];
       buf[0] = '0';
       len = strlen(buf);
       }
    for (i = 0; i < len; i++) {      if (i&1) /* odd is lower*/
          packed[i/2] |= (buf[i]&0x0f);
       else
          packed[i/2] = (char)((buf[i]&0x0f)<<4);
       }
    packed[i/2] |= negative ? 13 : 12;
    return (i/2)+1;   /* return length of packed decimal string*/}
/* *  Convert packed decimal to real
 *
 * packed: pointer to input packed decimal string
 * result: pointer to output result real*/
int routine cvtpdr(real *result, char *packed, int digits)
{  char string[32+1];   /* maximum packed decimal length*/
    int i, n;
     /*     * Convert packed decimal to leading separate
   */
    cvtps(string, packed);
    /*  * set packed decimal precision to 'digits' fractional digits*/
    for (i = 0; string[i]; i++);
    for (n = 0; n < digits && i >= 0; --i, n++)
       string[i+1] = string[i];
    string[i+1] = '.';
    /*  * Convert ascii string to real*/
    return ator(result, string);}
/* *  Convert real to packed decimal
 *
 * packed: pointer to output packed decimal string
 * digits: sizeof output buffer (as packed decimal digits)
 * val:    pointer to real value to convert*/
int routine cvtrpd (char *packed, int digits, int precision, real *val)
{  realinfo info;
    char buf[32+1];      /* maximum size of a packed decimal field*/
    char pbuf[32+1];     /* packed buffer*/
    int p_digits;
    DESC_S desc;
    int i;
    set$desc (&desc, sizeof(buf), buf);
    /*  * Convert real to ascii string*/
    rtoa(val, &desc, precision, &info, 0);
    /*  * Convert ascii string to packed decimal*/
    p_digits = cvtsp(pbuf, buf, 0);
    digits = (digits/2)+1;     /* convert digits to bytes + sign*/
    /*  * fill packed decimal string with required
     * leading packed zeros.*/
    for (i = p_digits; i < digits; i++)
       *packed++ = '\0';
    /*  * now fill with converted packed number*/
    for (i = 0; i < p_digits; i++)
       *packed++ = pbuf[i];
```

```
   return SUCCESS;}
/* *  cvtlp -- convert long to packed decimal
 **/
int routine cvtlp(int *expr, int digits, char *buf)
{  real r = *expr;
   return cvtrpd(buf, digits, 0, &r);}
/* *  cvtpl -- convert packed decimal to long
 **/
int routine cvtpl(int digits, char *buf, int *result)
{  int stat;
   real r;
   stat = cvtpdr(&r, buf, digits);
   *result = (int)r.ip;
   return stat;}
```

DEBUG DUMP HEX BYTES

```
/////////////////////////////////////////////////////////////////////////////////
// DHEXB DUMP HEX BYTES OF A MEMORY LOCATION
/////////////////////////////////////////////////////////////////////////////////
// Brief description: This function is a debug function. It will dump in hexadecimal
//  format with ascii characters at the right, a block a memory
//  starting at address 'a' for bytes 'n'. Unprintable character
//  hex values are displayed but their ascii value is '.'.
//  This function is called when debugging the math routines
//  at a very low level. It is called to dump the contents of
//  'real' 64-bit integers (and their 128-bit intermediate values)
//  to check that the hex values are indeed correct.
//  Any block of memory may be dumped with this function; not
//  just 'real' 64-bit (or intermediate 128-bit) values. I have
//  used this function to dump 'pcode' blocks of memory when I
//  was not sure what was being generated.
// Expected:  a          =   the start address (any type)
//            n          =   number of bytes to dump.
//   Result:  return     =   void
/////////////////////////////////////////////////////////////////////////////////
void routine dhexb (void *a, int n)
{  unsigned char *s = (unsigned char *) a;
   char msg[256];
   char buf[32];
   int sn = n;
   int i;
   for (; n > 0; n -= i)
     {         sprintf (msg, "  %04d  ", sn - n);
        for (i = 0; i < 16 && i < n; i++)
         {           sprintf (buf, "%02X ", s[i]);
           strcat (msg, buf);         }
        for (; i < 16; i++)
          strcat (msg, "    ");
        strcat (msg, "  ");
        for (i = 0; i < 16 && i < n; i++, s++)
         {           sprintf (buf, "%c", *s < ' ' || *s > '~' ? '.' : *s);
           strcat (msg, buf);         }
        logf ("%s\n", msg);
     }}
/////////////////////////////////////////////////////////////////////////////////
// Brief description: add 2 reals and place result in a third
// Expected:  1st arg    =   unused
//            a          =   the address of a quadword (integer pointer)
//            b          =   the address of a quadword (integer pointer)
//            c          =   the address of a quadword (integer pointer)
//
//   Result:  c          = a + b;
/////////////////////////////////////////////////////////////////////////////////
void routine addm (int, int *a, int *b, int *c)
{  *(int64 *) c = *(int64 *) a + *(int64 *) b;}
```

MORE MATH AND COMPARE FUNCTIONS

```
/////////////////////////////////////////////////////////////////////////////////
// SUBM SUBTRACT 2 QUADWORDS AND PLACE RESULT IN A THIRD
/////////////////////////////////////////////////////////////////////////////////
// Brief description: subtract 2 reals and place result in a third
// Expected:  1st arg    =   unused
//            a          =   the address of a quadword (integer pointer)
```

```
//           b          =   the address of a quadword (integer pointer)
//           c          =   the address of a quadword (integer pointer)
//
//   Result:  c          = b - a;
//////////////////////////////////////////////////////////////////////////
void routine subm (int, int *a, int *b, int *c)
{  *(int64 *) c = *(int64 *) b - *(int64 *) a;}
//////////////////////////////////////////////////////////////////////////
```

// IS_NEGATIVE IS A REAL NEGATIVE

```
//////////////////////////////////////////////////////////////////////////
// Brief description: test the sign bit of a real
// Expected:    a   =   pointer to a real
//   Result:    true if negative
//////////////////////////////////////////////////////////////////////////
inline int routine is_negative_real (real * a)
{  if (a->ip)
      return a->ip < 0;
   else
      return a->fp < 0;}
//////////////////////////////////////////////////////////////////////////
```

// IS_NEGATIVE IS A 64-BIT INTEGER ARRAY NEGATIVE

```
//////////////////////////////////////////////////////////////////////////
// Brief description: test the sign bit of a 64-bit integer array
// Expected:    a   =   pointer to start of a 64-bit integer array
//                      (or a real).
//              n   =   length of array (number of 64-bit integers)
//   Result:    true if negative
//////////////////////////////////////////////////////////////////////////
inline int routine is_negative (void *a, int n)
{  return (int) ((((uint64 *) a)[n - 1] >> 63) & 1);  /* shift the 64-bit integer down*/}
//////////////////////////////////////////////////////////////////////////
```

// IS A REAL LESS THAN ANOTHER

```
//////////////////////////////////////////////////////////////////////////
// Brief description: is a < b
// Expected:    a   =   pointer to a real
//              b   =   pointer to a real
//   Result:    result a < b
//////////////////////////////////////////////////////////////////////////
inline int lss (real * a, real * b)
{  if (a->ip < b->ip)
      return 1;
   if (a->ip > b->ip)
      return 0;
   return a->fp < b->fp;}
//////////////////////////////////////////////////////////////////////////
```

// NEGATE A 64-BIT INTEGER ARRAY

```
//////////////////////////////////////////////////////////////////////////
// Brief description: negate a 64-bit integer array (make negative positive or
//  positive negative)
// Expected:    a   =   pointer to start of a 64-bit integer array
//                      (or a real).
//              n   =   length of array (number of 64-bit integers)
//   Result:    void
//////////////////////////////////////////////////////////////////////////
inline void routine negate (void *a, int n)
{  int i;
   ((int64 *) a)[0] = -((int64 *) a)[0];  /* two's compliment*/
   for (i = 1; i < n; i++)
      ((int64 *) a)[i] = ~((int64 *) a)[i];  /* one's complement*/}
//////////////////////////////////////////////////////////////////////////
```

// COPY A 64-BIT INTEGER ARRAY

```
//////////////////////////////////////////////////////////////////////////
// Brief description: inline copy a number of 64-bit integers from 'b' to 'a'
//  (the pointers can point at an integer array or a real)
// Expected:    a   =   pointer to start of a 64-bit integer array
//                      (or a real).
//              b   =   pointer to start of a 64-bit integer array
//                      (or a real).
//              n   =   length of array (number of 64-bit integers)
//   Result:    void
//////////////////////////////////////////////////////////////////////////
inline void routine cpy (void *a, void *b, int n)
{  int i;
   for (i = 0; i < n; i++)
      ((int64 *) a)[i] = ((int64 *) b)[i];}
//////////////////////////////////////////////////////////////////////////
```

// SHIFTUP BY ONE BIT A 64-BIT INTEGER ARRAY

```
//////////////////////////////////////////////////////////////////////////
```

```
// Brief description: shiftup by one bit a 64-bit integers array
// (the pointers can point at an integer array or a real)
// Expected:     a    =   pointer to start of a 64-bit integer array
//                        (or a real).
//               n    =   length of array (number of 64-bit integers)
//   Result:     void
//////////////////////////////////////////////////////////////////////
inline void routine shiftup (void *a, int n)
{  int c, carry = 0;
   int i;
   for (i = 0; i < n; i++)
     {        c = (int) (((uint64 *) a)[i] >> 63) & 1;    /* shift the 64-bit integer down*/
       ((int64 *) a)[i] <<= 1;
       ((int64 *) a)[i] |= carry;
       carry = c;
     }}
//////////////////////////////////////////////////////////////////////
```

// SHIFTDOWN BY ONE BIT A 64-BIT INTEGER ARRAY

```
//////////////////////////////////////////////////////////////////////
// Brief description: shiftdown by one bit a 64-bit integers array
// (the pointers can point at an integer array or a real)
// Expected:     a    =   pointer to start of a 64-bit integer array
//                        (or a real).
//               n    =   length of array (number of 64-bit integers)
//   Result:     void
//////////////////////////////////////////////////////////////////////
inline void routine shiftdown (void *a, int n)
{  int64 c, carry = 0;
   int i;
   for (i = n - 1; i >= 0; --i)
     {        c = (int) (((uint64 *) a)[i]) & 1;
       ((uint64 *) a)[i] >>= 1; /* unsigned shift down*/
       ((uint64 *) a)[i] |= (carry << 63);
       carry = c;
     }}
```

// LSS COMPARE FUNCTIONS FOR AN ARRAY OF 64-BIT INTEGERS

```
//////////////////////////////////////////////////////////////////////
// LSS LESS THAN COMPARE AN ARRAY OF 64-BIT INTEGERS
//////////////////////////////////////////////////////////////////////
// Expected:
//             a    =   address of 64-bit integer array
//             b    =   address of 64-bit integer array
//             n    =   number of 64-bit integers
//   Result:  TRUE if (a < b) else FALSE;
//////////////////////////////////////////////////////////////////////
inline int routine lss (void *a, void *b, int n)
{  int i;
   if (((int64 *) a)[n - 1] != ((int64 *) b)[n - 1])
     return ((int64 *) a)[n - 1] < ((int64 *) b)[n - 1];    /* signed compare*/
   for (i = n - 1; i >= 0; --i)
     {        if (((int64 *) a)[i] != ((int64 *) b)[i])
         return ((uint64 *) a)[i] < ((uint64 *) b)[i]; /* unsigned compare*/       }
   return 0;               /* false equal*/}
//////////////////////////////////////////////////////////////////////
```

// LEQ LESS THAN OR EQUAL COMPARE TWO REALS

```
//////////////////////////////////////////////////////////////////////
// Expected:
//             a    =   address of real
//             b    =   address of real
//   Result:  TRUE if (a <= b) else FALSE;
//////////////////////////////////////////////////////////////////////
inline int routine leq (real * a, real * b)
{  if (a->ip <= b->ip)
     return 1;
   if (a->ip > b->ip)
     return 0;
   return a->fp <= b->fp;}
//////////////////////////////////////////////////////////////////////
```

// LEQ LESS THAN OR EQUAL COMPARE AN ARRAY OF 64-BIT INTEGERS

```
//////////////////////////////////////////////////////////////////////
// Expected:
//             a    =   address of 64-bit integer array
//             b    =   address of 64-bit integer array
//             n    =   number of 64-bit integers
//   Result:  TRUE if (a <= b) else FALSE;
//////////////////////////////////////////////////////////////////////
```

```
inline int routine leq (void *a, void *b, int n)
{  int i;
   if (((int64 *) a)[n - 1] != ((int64 *) b)[n - 1])
      return ((int64 *) a)[n - 1] < ((int64 *) b)[n - 1];   /* signed compare*/
   for (i = n - 1; i >= 0; --i)
     {         if (((int64 *) a)[i] != ((int64 *) b)[i])
           return ((uint64 *) a)[i] < ((uint64 *) b)[i]; /* unsigned compare*/      }
   return 1;                /* true equal*/}
//////////////////////////////////////////////////////////////////////////////
// GEQ GREATER THAN OR EQUAL COMPARE AN ARRAY OF 64-BIT INTEGERS
//////////////////////////////////////////////////////////////////////////////
// Expected:
//            a   =   address of 64-bit integer array
//            b   =   address of 64-bit integer array
//            n   =   number of 64-bit integers
//   Result:  TRUE if (a >= b) else FALSE;
//////////////////////////////////////////////////////////////////////////////
inline int routine geq (void *a, void *b, int n)
{  int i;
   if (((int64 *) a)[n - 1] != ((int64 *) b)[n - 1])
      return ((int64 *) a)[n - 1] > ((int64 *) b)[n - 1];   /* signed compare*/
   for (i = n - 1; i >= 0; --i)
     {         if (((int64 *) a)[i] != ((int64 *) b)[i])
           return ((uint64 *) a)[i] > ((uint64 *) b)[i]; /* unsigned compare*/      }
   return 1;                /* true equal*/}
//////////////////////////////////////////////////////////////////////////////
// SET TO ZERO A 64-BIT INTEGER ARRAY
//////////////////////////////////////////////////////////////////////////////
// Brief description: set to zero a 64-bit integer array
//  (the pointers can point at an integer array or a real)
// Expected:     a   =   pointer to start of a 64-bit integer array
//                       (or a real).
//               n   =   length of array (number of 64-bit integers)
//   Result:    void
//////////////////////////////////////////////////////////////////////////////
inline void routine set_zero (void *a, int n)
{  int i;
   for (i = 0; i < n; i++)
      ((int64 *) a)[i] = 0;}
//////////////////////////////////////////////////////////////////////////////
// TEST IF A 64-BIT INTEGER ARRAY IS ZERO
//////////////////////////////////////////////////////////////////////////////
// Brief description: test if a 64-bit integer array is zero
//  (the pointers can point at an integer array or a real)
// Expected:     a   =   pointer to start of a 64-bit integer array
//                       (or a real).
//               n   =   length of array (number of 64-bit integers)
//   Result:    void
//////////////////////////////////////////////////////////////////////////////
inline int routine is_zero (void *a, int n)
{  int i;
   for (i = 0; i < n; i++)
      if (((int64 *) a)[i])
         return 0;
   return 1;                /* yes zero*/}
//////////////////////////////////////////////////////////////////////////////
// ADD64 ADD 2 64-BIT INTEGERS AND PLACE RESULT IN A THIRD
//////////////////////////////////////////////////////////////////////////////
// Brief description: add 2 reals and place result in a third
// Expected:
//            a          =   the address of a 64-bit real
//            b          =   the address of a 64-bit real
//            c          =   the address of a 64-bit real
//
//   Result:  c          = a + b;
//////////////////////////////////////////////////////////////////////////////
void routine add64 (real * a, real * b, real * c)
{  *(int64 *) c = *(int64 *) a + *(int64 *) b;}
//////////////////////////////////////////////////////////////////////////////
// SUB64 SUBTRACT 2 64-BIT INTEGERS AND PLACE RESULT IN A THIRD
//////////////////////////////////////////////////////////////////////////////
// Brief description: subtract 2 reals and place result in a third
// Expected:
//            a          =   the address of a 64-bit integer (void pointer)
//            b          =   the address of a 64-bit integer (void pointer)
//            c          =   the address of a 64-bit integer (void pointer)
//
```

```
//   Result:  c          = a - b;
//////////////////////////////////////////////////////////////////////////////
void routine sub64 (void *a, void *b, void *c)
{  *(int64 *) c = *(int64 *) a - *(int64 *) b;}
//////////////////////////////////////////////////////////////////////////////
```

// ADD A 64-BIT INTEGER ARRAY (LENGTH OF TWO)

```
//////////////////////////////////////////////////////////////////////////////
// Brief description: inline add a length of two 64-bit integers a = a + b
//  slightly faster for a 'normal' add of a 128-bit integer
//  (the pointers can point at an integer array or a real)
// Expected:     a    =   pointer to start of a 64-bit integer array
//                        (or a real).
//               b    =   pointer to start of a 64-bit integer array
//                        (or a real).
//   Result:    void
//////////////////////////////////////////////////////////////////////////////
inline void routine add2 (void *a, void *b)
{  uint64 was;
   int64 carry = 0;
   was = ((uint64 *) a)[0];
   ((uint64 *) a)[0] += ((uint64 *) b)[0];
   carry = (((uint64 *) a)[0] < was);
   ((uint64 *) a)[1] += ((uint64 *) b)[1] + carry;}
//////////////////////////////////////////////////////////////////////////////
```

// SUBTRACT A 64-BIT INTEGER ARRAY (LENGTH OF TWO)

```
//////////////////////////////////////////////////////////////////////////////
// Brief description: inline subtract a length of two 64-bit integers a = a - b
//  slightly faster for a 'normal' subtract of a 128-bit integer
//  (the pointers can point at an integer array or a real)
// Expected:     a    =   pointer to start of a 64-bit integer array
//                        (or a real).
//               b    =   pointer to start of a 64-bit integer array
//                        (or a real).
//   Result:    void
//////////////////////////////////////////////////////////////////////////////
inline void routine sub2 (void *a, void *b)
{  uint64 was;
   int64 carry = 0;
   was = ((uint64 *) a)[0];
   ((uint64 *) a)[0] -= ((uint64 *) b)[0];
   carry = (((uint64 *) a)[0] > was);
   ((uint64 *) a)[1] -= ((uint64 *) b)[1] + carry;}
//////////////////////////////////////////////////////////////////////////////
```

// SCALEUP A 128-BIT INTEGER BY SCALE

```
//////////////////////////////////////////////////////////////////////////////
// Brief description: Multiply the 128-bit integer by SCALE
// Expected:
//             _a  =   address of 64-bit integer array (_a * _b)
//   Result:  void
//////////////////////////////////////////////////////////////////////////////
inline void routine scaleup (void *_a)
{  int64 a[] = { 0, 0 };
   int64 b[] = { SCALE, 0 };
   int64 c[] = { 0, 0 };
   int na = 0;
   int i;
   cpy (a, _a, 2);
   if (is_negative (a, 2))
     {                        /* negative*/
        negate (a, 2);        /* make positive*/
        na = 1;       }
   if (lss (a, b, 2))
     {                        /* swap*/
        int64 temp[2];
        cpy (temp, b, 2);
        cpy (b, a, 2);
        cpy (a, temp, 2);      }
   for (i = 0; i < 64 && (b[0] || b[1]); i++)
     {         if (b[0] & 1)
           add2 (c, a);
        shiftdown (b, 2);
        shiftup (a, 2);     }
   if (na == 1)
     {         negate (c, 2);      /* make positive*/       }
   cpy (_a, c, 2);}
//////////////////////////////////////////////////////////////////////////////
```

```
// SUBTRACT A 64-BIT INTEGER ARRAY
////////////////////////////////////////////////////////////////////////////////
// Brief description: inline subtract a number of 64-bit integers a = a - b
//  (the pointers can point at an integer array or a real)
// Expected:    a    =   pointer to start of a 64-bit integer array
//                        (or a real).
//              b    =   pointer to start of a 64-bit integer array
//                        (or a real).
//              n    =   length of array (number of 64-bit integers)
//   Result:    void
////////////////////////////////////////////////////////////////////////////////
inline void routine sub (void *a, void *b, int n)
{  uint64 was;
   int64 carry = 0;
   int i;
   for (i = 0; i < n; i++)
     {         was = ((uint64 *) a)[i];
        ((uint64 *) a)[i] -= ((uint64 *) b)[i] + carry;
        carry = (((uint64 *) a)[i] > was);
     }}
////////////////////////////////////////////////////////////////////////////////
// ADD A 64-BIT INTEGER ARRAY
////////////////////////////////////////////////////////////////////////////////
// Brief description: inline add a number of 64-bit integers a = a + b
//  (the pointers can point at an integer array or a real)
// Expected:    a    =   pointer to start of a 64-bit integer array
//                        (or a real).
//              b    =   pointer to start of a 64-bit integer array
//                        (or a real).
//              n    =   length of array (number of 64-bit integers)
//   Result:    void
////////////////////////////////////////////////////////////////////////////////
inline void routine add (void *a, void *b, int n)
{  uint64 was;
   int64 carry = 0;
   int i;
   for (i = 0; i < n; i++)
     {         was = ((uint64 *) a)[i];
        ((uint64 *) a)[i] += ((uint64 *) b)[i] + carry;
        carry = (((uint64 *) a)[i] < was);
     }}
////////////////////////////////////////////////////////////////////////////////
// SCALEDOWN A 128-BIT INTEGER BY SCALE
////////////////////////////////////////////////////////////////////////////////
// Expected:
//              _a   =   address of 64-bit integer array
//   Result:    void
////////////////////////////////////////////////////////////////////////////////
inline void routine scaledown (void *_a, int n)
{  uint64 a[] = { 0, 0, 0, 0 };
   uint64 r[] = { 0, 0 };
   uint64 scale[] = { SCALE, 0, 0, 0 };    /* SCALE must be less than an int64*/
   uint64 b[] = { SCALE, 0, 0, 0 };
   uint64 pow[] = { 1, 0 };
   int na = 0;
   int i;
   cpy (a, _a, n);
   if (!a[1] && !a[2] && !a[3])
     {         a[0] /= SCALE;
        cpy (_a, a, n);
        return;       }
   if (is_negative (a, n))
     {                      /* negative*/
        negate (a, n);
        na = 1;       }
   /* shift 'b' until it is greater than 'a'*/
   for (i = 0; i < 128 && leq (b, a, n); i += 2)
     {         shiftup (b, n);
        shiftup (pow, 2);
        shiftup (b, n);
        shiftup (pow, 2);      }
   if (is_zero (pow, 2))
     {         runtime$error (MSG_FLOATOVF);  /* overflow*/      }
   while (leq (scale, a, n))
```

```
    {                       /* while still a divisor*/
       while (lss (a, b, n))
        {            shiftdown (b, n);
           shiftdown (pow, 2);         }
       if (!a[1] && !a[2] && !a[3])
        {                /* down to a 64-bit integer?*/
           a[0] /= SCALE;
           add2 (r, a);
           break;            }
       sub (a, b, n);       /* subtract largest power*/
       add2 (r, pow);       /* increment result*/       }
  if (na == 1)
    {       negate (r, 2);      }
  cpy (_a, r, 2);}
```

// REALIP GET INTEGRAL PORTION OF A REAL

```
// Brief description: get the integral portion of a real removing the fractional part.
// Expected:
//            in         =   address of input real
//            out        =   the address of a 64-bit real (void pointer)
//   Result:  void       :   out = integral portion of in
void routine realip (real * in, real * out)
{  out->fp = 0;                /* no fractional part*/
   out->ip = in->ip;        /* just the integral part*/}
```

// WHOLE_INT GET INTEGRAL PORTION OF A REAL

```
// Brief description: get the integral portion of a real removing the fractional part.
// Expected:
//            in         =   address of input real
//            out        =   the address of a 64-bit real (void pointer)
//   Result:  void       :   out = integral portion of in
void routine whole_int (real *in, real *out)
{  realip(in, out);}
```

// CVTLR CONVERT LONG TO A REAL

```
// Brief description: convert an integer to a 64-bit real
// Expected:
//            a          =   input integer
//            b          =   the address of a 64-bit real
//   Result:  1          :   b = a
int routine cvtlr (int _a, real * b)
{  real a = _a;
   *b = a;
   return 1;}
```

// CONVERT REAL TO A LONG

```
// Brief description: convert a real to a long with rounding
// Expected:
//            a        =   address of input 64-bit real
//            b        =   address of output integer
//   Result:  1        :   a = rounded real b
int routine cvtrl (real *r, int *i)
{   *i = (int) r->ip;               // get integer part
   if (r->fp != 0                  // Non-ZERO fraction?
      && (r->fp >= SCALE / 2       // and we have to round it
      ||  r->fp <= -(SCALE/2)))
      {     if (*i < 0)                  // round down if negative
         *i = *i - 1;
      else
          *i = *i + 1;              // round UP if zero or positive
      }
   return 1;}
```

// GETINT GET AN INTEGER FROM A REAL

```
// Brief description: Get an integer from a real
// Expected:
//            n      =   reference to a real
//   Result:  integer value
```

```
//////////////////////////////////////////////////////////////////////
int routine getint (real & n)
{  int i;
   cvtrl (&n, &i);
   return i;}
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// I P
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Brief description: Truncates a real number at the decimal point and
//  returns the integer portion.
// Expected:  in    -  passed real number
//   Result:  out   - the integer portion of a real number
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
void routine ip (int *out, real * in)
{  *out = (int) in->ip;     /* truncate a int64 down to an int*/}                   // ip
//////////////////////////////////////////////////////////////////////
```

// CMP64 COMPARE TWO 64-BIT INTEGERS

```
//////////////////////////////////////////////////////////////////////
// Brief description: compare two 64-bit integers
// Expected:
//            a        =    address of 64-bit integer
//            b        =    address of 64-bit integer
//   Result:  0  = equal
//            -1 = less than
//            1  = equal
//////////////////////////////////////////////////////////////////////
int routine cmp64 (void *a, void *b)
{  if (*(int64 *) a == *(int64 *) b)
      return 0;
   if (*(int64 *) a < *(int64 *) b)
      return -1;
   return 1;}
//////////////////////////////////////////////////////////////////////
```

// DHEX64 DUMP A 64-BIT INTEGER IN HEX

```
//////////////////////////////////////////////////////////////////////
// Brief description: Dump a 64-bit integer in hex. This is a debug tool
//  for 64-bit arithmetic.
// Expected:
//            buf      =    dump hex characters to this buffer
//            _s       =    start address of 64-bit quadword
//   Result:  void
//////////////////////////////////////////////////////////////////////
static void routine dhex64 (char *buf, void *_s)
{  unsigned int *s = (unsigned int *) _s;
   sprintf (buf, "%08X%08X", s[1], s[0]);}
//////////////////////////////////////////////////////////////////////
```

// DHEX96 DUMP A 96-BIT INTEGER IN HEX

```
//////////////////////////////////////////////////////////////////////
// Brief description: Dump a 96-bit integer in hex. This is a debug tool
//  for 96-bit arithmetic.
// Expected:
//            buf      =    dump hex characters to this buffer
//            _s       =    start address of 96-bit quadword
//   Result:  void
//////////////////////////////////////////////////////////////////////
void routine dhex96 (char *buf, void *_s)
{  unsigned int *s = (unsigned int *) _s;
   dhex64 (buf, s + 1);
   sprintf (buf + 16, "%08X", s[0]);}
//////////////////////////////////////////////////////////////////////
```

// DHEX128 DUMP A 128-BIT INTEGER IN HEX

```
//////////////////////////////////////////////////////////////////////
// Brief description: Dump a 128-bit integer in hex. This is a debug tool
//  for 128-bit arithmetic.
// Expected:
//            buf      =    dump hex characters to this buffer
//            _s       =    start address of 128-bit quadword
//   Result:  void
//////////////////////////////////////////////////////////////////////
void routine dhex128 (char *buf, void *_s)
{  unsigned int *s = (unsigned int *) _s;
   dhex64 (buf, s + 2);
   dhex64 (buf + 16, s);}
//////////////////////////////////////////////////////////////////////
```

// RTOD CONVERT REAL TO DOUBLE

```
//////////////////////////////////////////////////////////////////////
// Brief description: Convert a real to a double.
```

```
// Expected:
//           a      =   address of real
//           b      =   address of double
//   Result:  void
////////////////////////////////////////////////////////////////
void routine rtod (real * a, double *b)
{  double d = a->fp;
   d /= SCALE;
   d += a->ip;
   *b = d;}
////////////////////////////////////////////////////////////////
```

// GETINT GET A DOUBLE FROM A REAL

```
////////////////////////////////////////////////////////////////
// Brief description: Get a double from a real
// Expected:
//           n      =   address of a real
//   Result:  double value
////////////////////////////////////////////////////////////////
double routine getdouble (real * n)
{  double d;
   rtod (n, &d);
   return d;}
////////////////////////////////////////////////////////////////
```

// DIV128 DIVIDE TWO 128-BIT INTEGERS

```
////////////////////////////////////////////////////////////////
// Brief description: Divide two 128-bit integers. This is used for intermediate
//   arithmetic.
// Expected:
//           _dvend  =   address of real dividend
//           _dvsor  =   address of real divisor
//           quote   =   address of real quotent (division result)
//           rem     =   address of real remainder
//   Result:  dividend/divisor = quotent, remainder
////////////////////////////////////////////////////////////////
void div128 (void *_dvend, void *_dvsor, void *_quote, void *rem)
{  uint64 dvend[] = { 0, 0 };
   uint64 sdvsor[] = { 0, 0 };
   uint64 dvsor[] = { 0, 0 };
   uint64 quote[] = { 0, 0 };
   uint64 pow[] = { 1, 0 };
   int i;
   cpy (dvend, _dvend, 2);    /* get local copy of dvend*/
   cpy (dvsor, _dvsor, 2);    /* get local copy of dvsor*/
   cpy (sdvsor, _dvsor, 2);   /* get local copy of dvsor*/
   if (is_zero (dvsor, 2))
     {                      /* no divisor ?*/
       runtime$error (MSG_DIVBY0); /* divide by zero*/      }
   /*  *  shift divisor until it is >= to dividend
    *  that is, whilst the divisor is less than the dividend.*/
   for (i = 0; i < 128 && lss (dvsor, dvend, 2); i += 2)
     {       shiftup (dvsor, 2);
       shiftup (pow, 2);
       shiftup (dvsor, 2);   /* shiftup twice to speed up*/
       shiftup (pow, 2);     }
   if (is_zero (pow, 2))
     {       runtime$error (MSG_FLOATOVF);  /* overflow*/      }
   /*  *  Here we subtract the largest possible divisor value
    *  from the divident each time adding power to the quotent.
    *  We are trying to reduce the divident as quickly as we can.*/
   while (leq (sdvsor, dvend, 2))
     {                      /* while still a divisor*/
       while (lss (dvend, dvsor, 2))
         {               /* follow the divident down in value*/
           shiftdown (dvsor, 2);
           shiftdown (pow, 2);        }
       if (!sdvsor[1] && !dvend[1])
         {               /* down to just 64-bit integers ?*/
           uint64 n[] = { 0, 0 };
           n[0] = dvend[0] / sdvsor[0];
           dvend[0] -= n[0] * sdvsor[0]; /* remainder*/
           add2 (quote, n);
           break;            }
       sub2 (dvend, dvsor);  /* subtract largest dvsor*/
       add2 (quote, pow); /* increment quotent*/     }
```

```
   if (rem)
      cpy (rem, dvend, 2); /* remainder*/
   cpy (_quote, quote, 2);    /* return result*/}
```

// MUL128 MULTIPLY TWO 128-BIT INTEGERS

```
// Brief description: Multiply two 128-bit integers. This is used for intermediate
//   arithmetic.
// Expected:
//             _a  =   address of 64-bit integer array (_a * _b)
//             _b  =   address of 64-bit integer array (_a * _b)
//             r   =   address of 64-bit integer array (result)
//   Result:  void
void mul128 (void *_a, void *_b, void *_c)
{  uint64 a[] = { 0, 0, 0, 0 };
   uint64 b[] = { 0, 0, 0, 0 };
   uint64 c[] = { 0, 0, 0, 0 };
   int i;
   cpy (a, _a, 2);
   cpy (b, _b, 2);
   /*  *  make 'b' the smaller of the two*/
   if (lss (a, b, 4))
     {        int64 temp[4];
        cpy (temp, a, 4);
        cpy (a, b, 4);
        cpy (b, temp, 4);       }
   for (i = 0; i < 128 && !is_zero (b, 4); i++)
     {
        if (b[0] & 1)
           add (c, a, 4);
        shiftdown (b, 4);      /* --> shift down*/
        shiftup (a, 4);     /* <-- shift up*/     }
   cpy (_c, c, 2);}
```

// DIVR DIVIDE TWO 128-BIT REALS

```
// Brief description: Divide two 128-bit integers. This is used for intermediate
//   arithmetic.
// Expected:
//             _dvend  =   address of real dividend
//             _dvsor  =   address of real divisor
//             quote   =   address of real quotent (division result)
//             rem     =   address of real remainder
//   Result:  dividend/divisor = quotent, remainder
void divr (real * dvend, real * dvsor, real * quote, real * rem)
{  real q;
   r96divide(dvend, dvsor, &q);
   if (rem) *rem = 0;
   *quote = q;}
```

// REAL::OPERATOR-() NEGATE A REAL

```
// Brief description: negate the current real as in -real
// Expected:    The current real class
//   Result:    real
real real::operator - ()
{  real r = *this;
   negate_real (&r);
   return r;}
```

// REAL::OPERATOR+=() ADD A REAL

```
// Brief description: add a real to ourselves as in real += real
// Expected:    The current real class
//   Result:    real reference
real &real::operator += (real r)
{                          /* real += real*/
   addr (this, &r);          /* add real*/
   return *this;}
```

// REAL::OPERATOR++() INCREMENT A REAL

```
// Brief description: increment a real by one as in real++
```

```
// Expected:    The current real class
//   Result:    real reference
//////////////////////////////////////////////////////////////////////
real &real::operator++ (int)
{                          /* real++*/
   ip++;
   return *this;}
//////////////////////////////////////////////////////////////////////
// REAL::OPERATOR-=() SUBTRACT A REAL
//////////////////////////////////////////////////////////////////////
// Brief description: subtract a real from ourselves as in real -= real
// Expected:    The current real class
//   Result:    real reference
//////////////////////////////////////////////////////////////////////
real &real::operator -= (real r)
{                          /* real -= real*/
   subr (this, &r);           /* subtract two reals*/
   return *this;}
//////////////////////////////////////////////////////////////////////
// REAL::OPERATOR--() DECREMENT A REAL
//////////////////////////////////////////////////////////////////////
// Brief description: decrement a real by one as in real--
// Expected:    The current real class
//   Result:    real reference
//////////////////////////////////////////////////////////////////////
real &real::operator-- (int)
{                          /* real--*/
   ip--;
   return *this;}
//////////////////////////////////////////////////////////////////////
// REAL::REAL() CONSTRUCT A REAL
//////////////////////////////////////////////////////////////////////
// Brief description: construct a real and initialize it to zero
// Expected:    The current real class
//   Result:    real reference
//////////////////////////////////////////////////////////////////////
real::real ()
{  ip = 0;
   fp = 0;}
//////////////////////////////////////////////////////////////////////
// REAL::REAL() CONSTRUCT A REAL FROM AN INTEGER
//////////////////////////////////////////////////////////////////////
// Brief description: construct a real from an integer
// Expected:    The current real class
//   Result:    real reference
//////////////////////////////////////////////////////////////////////
real::real (int i)
{                          /* construct a real from an integer*/
   ip = i;
   fp = 0;}
//////////////////////////////////////////////////////////////////////
// REAL::REAL() CONSTRUCT A REAL FROM A 64-BIT INTEGER
//////////////////////////////////////////////////////////////////////
// Brief description: construct a real from a 64-bit integer
// Expected:    The current real class
//   Result:    real reference
//////////////////////////////////////////////////////////////////////
real::real (int64 i)
{                          /* construct a real from a 64-bit integer*/
   ip = i;
   fp = 0;}
//////////////////////////////////////////////////////////////////////
// REAL::REAL() CONSTRUCT A REAL FROM A DOUBLE
//////////////////////////////////////////////////////////////////////
// Brief description: construct a real from a double
// Expected:    The current real class
//   Result:    real reference
//////////////////////////////////////////////////////////////////////
real::real (double d)
{  dtor (&d, this);}
//////////////////////////////////////////////////////////////////////
//  real (whole, fraction)
//////////////////////////////////////////////////////////////////////
real::real (int64 whole, int fract)
{      ip = whole;
       fp = fract;}
//////////////////////////////////////////////////////////////////////
// REAL::OPERATOR-() SUBTRACT A REAL FROM ANOTHER REAL
//////////////////////////////////////////////////////////////////////
// Brief description: subtract a real from another real
```

```
// Expected:    The current real class
//   Result:    real reference
////////////////////////////////////////////////////////////////////////////////
real real::operator - (real i)
{                        /* real - real*/
   real r = *this;
   return r -= i;}
////////////////////////////////////////////////////////////////////////////////
// REAL::OPERATOR+() ADD A REAL TO ANOTHER REAL
////////////////////////////////////////////////////////////////////////////////
// Brief description: add a real to another real
// Expected:    The current real class
//   Result:    real reference
////////////////////////////////////////////////////////////////////////////////
real real::operator + (real i)
{                        /* real + real*/
   real r = *this;
   return r += i;}
////////////////////////////////////////////////////////////////////////////////
// REAL::OPERATOR<() COMPARE TWO REALS FOR LESS THAN
////////////////////////////////////////////////////////////////////////////////
// Brief description: compare two reals for less than
// Expected:    The current real class
//   Result:    real reference
////////////////////////////////////////////////////////////////////////////////
int real::operator < (real i)
{                        /* real < real*/
   return lss (this, &i);}
////////////////////////////////////////////////////////////////////////////////
// REAL::OPERATOR<=() COMPARE TWO REALS FOR LESS THAN OR EQUAL
////////////////////////////////////////////////////////////////////////////////
// Brief description: compare two reals for less than or equal
// Expected:    The current real class
//   Result:    real reference
////////////////////////////////////////////////////////////////////////////////
int real::operator <= (real i)
{                        /* real <= real*/
   return leq (this, &i);}
////////////////////////////////////////////////////////////////////////////////
// REAL::OPERATOR<=() COMPARE TWO REALS FOR GREATER THAN
////////////////////////////////////////////////////////////////////////////////
// Brief description: compare two reals for greater than
// Expected:    The current real class
//   Result:    real reference
////////////////////////////////////////////////////////////////////////////////
int real::operator > (real i)
{                        /* real > real*/
   return lss (&i, this);     /* if 'i' is less then we are greater*/}
////////////////////////////////////////////////////////////////////////////////
// REAL::OPERATOR>=() COMPARE TWO REALS FOR GREATER THAN OR EQUAL
////////////////////////////////////////////////////////////////////////////////
// Brief description: compare two reals for greater than or equal
// Expected:    The current real class
//   Result:    real reference
////////////////////////////////////////////////////////////////////////////////
int real::operator >= (real i)
{                        /* real >= real*/
   if (lss (this, &i))        /* we are less*/
      return 0;
   return 1;                  /* we must be >=*/}
////////////////////////////////////////////////////////////////////////////////
// REAL::OPERATOR%() THE MODULUS REMAINDER OF A REAL AND AN INTEGER
////////////////////////////////////////////////////////////////////////////////
// Brief description: the modulus remainder of a real and an integer
// Expected:    The current real class
//   Result:    real reference
////////////////////////////////////////////////////////////////////////////////
real real::operator % (int i)
{  real a = (*this / (real) i);
   uint64 scale[] = { SCALE, 0 };
   div128 (this, (real *) scale, this, 0);
   real r = *this - (real) (a * (int64) i);
   return r;}
////////////////////////////////////////////////////////////////////////////////
// REAL::OPERATOR*() MULTIPLY TWO REALS
////////////////////////////////////////////////////////////////////////////////
// Brief description: multiply two reals
// Expected:    The current real class
//   Result:    real reference
////////////////////////////////////////////////////////////////////////////////
```

```
real real::operator * (real m)
{  real r;
   mulr (this, &m, &r);
   return r;}
/////////////////////////////////////////////////////////////////////
// REAL::OPERATOR&() A BIT AND OF A REAL AND AN INTEGER
/////////////////////////////////////////////////////////////////////
// Brief description: a bit and of a real and an integer
// Expected:    The current real class
//   Result:    real reference
/////////////////////////////////////////////////////////////////////
int real::operator & (int m)
{  return (int) (ip & m);     /* bitwise AND on integer portion only*/}
/////////////////////////////////////////////////////////////////////
// REAL::OPERATOR/() DIVIDE TWO REALS
/////////////////////////////////////////////////////////////////////
// Brief description: divide two reals
// Expected:    The current real class
//   Result:    real reference
/////////////////////////////////////////////////////////////////////
real real::operator / (real m)
{  real r;
   divr (this, &m, &r);
   return r;}
/////////////////////////////////////////////////////////////////////
// REAL::OPERATOR*=() MULTIPLY TWO REALS AND ASSIGN
/////////////////////////////////////////////////////////////////////
// Brief description: multiply two reals and assign
// Expected:    The current real class
//   Result:    real reference
/////////////////////////////////////////////////////////////////////
real real::operator *= (real r)
{  return *this = (*this * r);}
/////////////////////////////////////////////////////////////////////
// REAL::OPERATOR/=() DIVIDE TWO REALS AND ASSIGN
/////////////////////////////////////////////////////////////////////
// Brief description: divide two reals and assign
// Expected:    The current real class
//   Result:    real reference
/////////////////////////////////////////////////////////////////////
real real::operator /= (real r)
{  return *this = (*this / r);}
/////////////////////////////////////////////////////////////////////
// CMPR() COMPARE TWO REALS FOR EQUALITY
/////////////////////////////////////////////////////////////////////
// Expected:    The address of two reals
//   Result:    -1 if a < b
//              0  if a == b
//              1  if a > b
/////////////////////////////////////////////////////////////////////
int cmpr (real * a, real * b)
{  if (*a == *b)
      return 0;
   if (*a < *b)
      return -1;
   return 1;}
/////////////////////////////////////////////////////////////////////
// REAL::OPERATOR==() COMPARE TWO REALS FOR EQUALITY
/////////////////////////////////////////////////////////////////////
// Expected:    The current real class
//   Result:    real reference
/////////////////////////////////////////////////////////////////////
int real::operator == (real r) { return ip == r.ip && fp == r.fp;}
/////////////////////////////////////////////////////////////////////
// REAL::OPERATOR!=() COMPARE TWO REALS FOR INEQUALITY
/////////////////////////////////////////////////////////////////////
// Expected:    The current real class
//   Result:    real reference
/////////////////////////////////////////////////////////////////////
int real::operator != (real r)
{  return !operator == (r);}
/////////////////////////////////////////////////////////////////////
// REAL::OPERATOR!() LOGICAL NOT OF A REAL (IS ZERO)
/////////////////////////////////////////////////////////////////////
// Expected:    The current real class
//   Result:    real reference
/////////////////////////////////////////////////////////////////////
int real::operator ! ()
{  return !ip && !fp;}
/////////////////////////////////////////////////////////////////////
```

```
// REAL::OPERATOR>>=() RIGHT BIT SHIFT OF A REAL AND ASSIGN
/////////////////////////////////////////////////////////////////////////////
// This function has not been implemented (it is not used)
// Expected:    The current real class
//   Result:    real reference
/////////////////////////////////////////////////////////////////////////////
real real::operator >>= (int)
{  return *this;}
/////////////////////////////////////////////////////////////////////////////
// REAL::OPERATOR>>=() LEFT BIT SHIFT OF A REAL AND ASSIGN
/////////////////////////////////////////////////////////////////////////////
// This function has not been implemented (it is not used)
// Expected:    The current real class
//   Result:    real reference
/////////////////////////////////////////////////////////////////////////////
real real::operator <<= (int)
{  return *this;}
/////////////////////////////////////////////////////////////////////////////
// REAL::OPERATOR>>=() LEFT BIT SHIFT OF A REAL
/////////////////////////////////////////////////////////////////////////////
// This function has not been implemented (it is not used)
// Expected:    The current real class
//   Result:    real reference
/////////////////////////////////////////////////////////////////////////////
real real::operator << (int i)
{  real r = *this << i;
   return r;}
/////////////////////////////////////////////////////////////////////////////
// REAL::OPERATOR>>=() RIGHT BIT SHIFT OF A REAL
/////////////////////////////////////////////////////////////////////////////
// This function has not been implemented (it is not used)
// Expected:    The current real class
//   Result:    real reference
/////////////////////////////////////////////////////////////////////////////
real real::operator >> (int i)
{  real r = *this >> i;
   return r;}
//  for rtoa() ascii digit table
#define TABLE_SCALE 10000
#define TABLE_DIGITS 4          /* number of '0's*/
static char ntab[TABLE_SCALE] [TABLE_DIGITS];
/////////////////////////////////////////////////////////////////////////////
//   MATH_INIT
/////////////////////////////////////////////////////////////////////////////
// Brief description: Initialize numeric table with ascii values
//  The table is initialized with the least significant bytes
//  first. For example, entry ntab[1] would be
//       ntab[1][0] = '1';       // least significant digit first
//       ntab[1][1] = '0';
//       ntab[1][2] = '0';
//       ntab[1][3] = '0';
//  and entry number ntab[1678] would be
//       ntab[1678][0] = '1';   // least significant digit first
//       ntab[1678][1] = '6';
//       ntab[1678][2] = '7';
//       ntab[1678][3] = '8';
//   Result:  void
/////////////////////////////////////////////////////////////////////////////
void routine math_init ()
{  char ascii_buf[32];
   char format_string[32];
   int i, n;
   sprintf (format_string, "%%0%dd", TABLE_DIGITS);    /* build format string*/
   for (i = 0; i < TABLE_SCALE; i++)
     {        sprintf (ascii_buf, format_string, i);  /* get ascii digits*/
        for (n = 0; n < TABLE_DIGITS; n++)    /* move those digits into table*/
           ntab[i][n] = ascii_buf[TABLE_DIGITS - (n + 1)];  /* least significant first*/
     }}
/////////////////////////////////////////////////////////////////////////////
// GET_ASCII_FRACTIONAL_DIGITS
/////////////////////////////////////////////////////////////////////////////
// Brief description: Get ascii fractional digits.
// Expected:
//             a        =   address a character array for ascii digits
//             i        =   index into above character array
//             n        =   binary integer to convert
//             p        =   scale fractional digits
```

```
//          digits  =   number of precision digits
//   Result:  i new index into a
/////////////////////////////////////////////////////////////////////////////////
inline int routine get_ascii_fractional_digits (char *a, int i, int n,
                                                int fract_digits, int digits)
{  int table_digit, index;
   if (n < 0)               /* calculate -1*/
      n = -n;               /* as 1*/
   while (fract_digits > 0)
     {         if (n < TABLE_SCALE)
         {               /* already less no need for a divide*/
            index = (int) n;  /* use the number as the index*/
            n = 0;          /* no more number*/        }
        else
         {           index = n % TABLE_SCALE;   /* remainder*/
            n /= TABLE_SCALE; /* new n*/        }
        for (table_digit = 0; table_digit < TABLE_DIGITS;
            table_digit++, --fract_digits)
           if (fract_digits > 0 && fract_digits <= digits)
              a[i++] = ntab[index][table_digit];       }
   return i;}                           /* get ascii fractional digits*/
/////////////////////////////////////////////////////////////////////////////////
```

// **GET_ASCII_WHOLE_DIGITS**

```
/////////////////////////////////////////////////////////////////////////////////
// Brief description: Get ascii whole integer digits.
// Expected:
//          a       =   address a character array for ascii digits
//          i       =   index into above character array
//          n       =   binary 64-bit integer to convert
//   Result:  i new index into a
/////////////////////////////////////////////////////////////////////////////////
inline int routine get_ascii_whole_digits (char *a, int i, real * _n)
{  int index, table_digit, skip_leading;
   int64 n = _n->ip;        /* integer portion*/
   do
     {         if (n < TABLE_SCALE)
         {               /* already less no need for a divide*/
            index = (int) n;  /* use the number as the index*/
            n = 0;          /* no more number*/        }
        else
         {           index = (int) (n % TABLE_SCALE); /* remainder*/
            n /= TABLE_SCALE; /* divide by TABLE_SCALE*/        }
        a[i++] = ntab[index][0]; /* always at least 1 character*/
        if (n)
         {               /* more to come*/
            for (table_digit = 1; table_digit < TABLE_DIGITS;
                table_digit++)
              a[i++] = ntab[index][table_digit];  /* convert all digits*/        }
        else
         {               /* skip leading '0's on last divide*/
            for (skip_leading = TABLE_DIGITS - 1; skip_leading;
                --skip_leading)
              if (ntab[index][skip_leading] != '0')
                {                for (table_digit = 1; table_digit <= skip_leading;
                      table_digit++)
                     a[i++] = ntab[index][table_digit];
                   break;
                }
         }     }
   while (n);
   return i;}                           /* get ascii whole digits*/
/////////////////////////////////////////////////////////////////////////////////
```

// **RTOA REAL TO ASCII**

```
/////////////////////////////////////////////////////////////////////////////////
// Brief description: Convert a real for ASCII output.
//  This routine is almost identical to rtoa...except that it fills
//  in realinfo with information about the converted number.  Information
//  includes stuff like:  whole_digits, fract_digits, neg or not, ...
// Expected:
//          in      =   address of 64-bit real
//          _desc   =   address of output descriptor
//          fdigits =   number of fractional digits to display
//          flags   =   (2 = implied decimal point) ??fix this??
//   Result:  real_info is filled
```

```
/////////////////////////////////////////////////////////////////////////////////
void routine rtoa (real * in, void *_desc, int fdigits, realinfo * rinfo,
            int flags)
{  DESC_S *desc = (DESC_S *) _desc;
   char *r = desc->dsc$a_pointer;
   int len = desc->dsc$w_length;
   int skip_trailing, i = 0;
   char buf[100];
   char *a = buf;
   real n;
   int f;
   int s = 0;
   int p;
   int is_implied_dec = (flags & 2);
   fill (sizeof (realinfo), rinfo, 0); /* zero all of the realinfo structure*/
   n = *in;
   if (is_negative_real (&n))
     {        negate_real (&n);
       s = 1;    }
   rinfo->neg = s;
   f = n.fp;               /* get fractional part*/
   /*  *  we build the string from left to right
    *  in reverse order fraction.whole digits*/
   if (fdigits || f != 0)
     {        /* fraction first right to left*/
       for (p = SCALE_DIGITS; p < fdigits; p++)
         a[i++] = '0';      /* fill with '0' digits first*/
       i = get_ascii_fractional_digits (a, i, f, SCALE_DIGITS, fdigits);
       for (skip_trailing = 0;
           skip_trailing < i && a[skip_trailing] == '0'; skip_trailing++);   /* skip '0's*/
       rinfo->fractdigits = i - skip_trailing; /* number of fractional digits ignoring trailing '0's*/
       if (!is_implied_dec)
         a[i++] = '.';      /* add a decimal point*/      }
   if (n.ip != 0)
     {                     /* get the whole ascii digits (all of them)*/
       int new_i = get_ascii_whole_digits (a, i, &n);
       rinfo->wholedigits = new_i - i;   /* number of whole digits*/
       i = new_i;      }
   if (s)
     if (i > 0)
        a[i++] = '-';
    rinfo->rlen = i;       /* length of number (relative to 1)*/
   a[i] = '\0';            // for easy debug
   for (p = 0; p < len - i; p++) // blank fill result
      r[p] = ' ';
   for (; p < len; p++)
     {                     // fill result with answer
       r[p] = a[--i];      // from reverse order
     }}                     //      rtoa
char* routine rtoa(real *n, int d)
{  static char buf[28];
   DESC_S desc[1];
   realinfo info;
   desc->dsc$a_pointer = buf;
   desc->dsc$w_length = sizeof (buf);
   rtoa (n, desc, d, &info, 0);  /* real to ascii*/
   return buf;}
/////////////////////////////////////////////////////////////////////////////////
```

// NAT_CNVOUT CONVERT A REAL FOR ASCII OUTPUT

```
/////////////////////////////////////////////////////////////////////////////////
// Brief description: Convert a real for ASCII output.
// Expected:
//           val      =   address of 64-bit real
//           result   =   address of ASCII buffer
//           digits   =   number of digits to display
//   Result:  status from math_cnvout
/////////////////////////////////////////////////////////////////////////////////
int routine math_cnvout (real * val, void *result, int digits)
{  DESC_S *desc = (DESC_S *) result;
   char *dest = desc->dsc$a_pointer;
   int rlen = desc->dsc$w_length;
   char tbuf[48];          //?? why 48
   int len = 38;           //?? why 38??
```

```
   int offset;
   DESC_S tmp (tbuf, len);
   realinfo rinfo;
   fill (rlen, dest, ' ');     // blank-fill result
   rtoa (val, &tmp, digits, &rinfo, 0);
   offset = len - rlen;     //?? what if RESULT buffer is bigger than ours?
   if (offset < 0)
      offset = 0;
   copy (rlen, &tbuf[offset], dest);
   return TRUE;}
///////////////////////////////////////////////////////////////////////////
// NEW_CNVOUT CONVERT A REAL FOR ASCII OUTPUT
///////////////////////////////////////////////////////////////////////////
// Brief description: Convert a real for ASCII output.
// Expected:
//             val      =   address of 64-bit real
//             result   =   address of ASCII buffer
//             digits   =   number of digits to display
//             flags    =   (2 = implied decimal point)  ??fix this to be a symbol??
//   Result:  realinfo structure is filled and returned
///////////////////////////////////////////////////////////////////////////
static int routine new_cnvout (real * val, void *result, int digits,
                              realinfo * rinfo, int flags, int max_len)
{  DESC_S *desc = (DESC_S *) result;
   char *dest = desc->dsc$a_pointer;
   int rlen = desc->dsc$w_length;
   char tbuf[48];            //?? why 48
   int len = 38;             //?? why 38??
   int offset;
   DESC_S tmp (tbuf, len);
   fill (rlen, dest, ' ');     // blank-fill result
   rtoa (val, &tmp, digits, rinfo, flags);
   offset = len - rlen;     //?? what if RESULT buffer is bigger than ours?
   if (offset < 0)
      offset = 0;
   if (max_len < rinfo->rlen) {     fill(rlen, dest, '*');
      return FALSE;
      }
   copy (rlen, &tbuf[offset], dest);
   return TRUE;}                         //  new_cnvout
const int cvtnumlen = 32;
static int ten_power[] = { 1,                   /* 0*/
   10,                       /* 1*/
   100,                   /* 2*/
   1000,                  /* 3*/
   10000,                    /* 4*/
   100000,                   /* 5*/
   1000000,               /* 6*/
   10000000,                 /* 7*/
   100000000,                /* 8*/
   1000000000                /* 9*/};
///////////////////////////////////////////////////////////////////////////
// SCALE$REAL SCALE A REAL TO A NUMBER OF FRACTIONAL DIGITS
///////////////////////////////////////////////////////////////////////////
// Brief description: Scale a real to a number of fractional digits
// Expected:
//             out      =   address of 64-bit real
//             in       =   address of 64-bit real
//             frac     =   number of fractional digits (scale to)
//   Result:  void
// Example:
//  scale 87.36 by 1 = 87.4
//  scale 87.36 by 0 = 87
//  scale 87.36 by -1 = 90
//  scale 87.36 by -2 = 100
///////////////////////////////////////////////////////////////////////////
void old$scale$real (real * out, real * in, int frac)
{  int64 scale[] = { SCALE, 0 };
   int64 one[] = { 1, 0 };
   int neg = 0;
   if (is_negative_real (in))
     {        negate_real (in);
        neg = 1;      }
```

```
   if (frac < -SCALE_DIGITS)
      frac = -SCALE_DIGITS;
   if (frac > SCALE_DIGITS)    // our maximum precision
      frac = SCALE_DIGITS;
   /*  *  here we calculate the scale*/
   if (frac < 0)
     {        /* -1 == 0.1*/
        int64 pow[] = { ten_power[-frac], 0 };
        mul128 (scale, pow, scale);     }
   else
     {        int64 pow[] = { ten_power[frac], 0 };
        div128 (scale, pow, scale, 0);     }
   if (!is_negative (scale, 2))
     {                       /* not negative*/
        int64 n[] = { 0, 0 }; /* whole 'n'*/
        int64 r[] = { 0, 0 }; /* remainder 'r'*/
        int64 a[] = { 0, 0 };
        int64 b[] = { 0, 0 };
        a[0] = in->ip;
        scaleup (a);
        b[0] = in->fp;
        add2 (a, b);        /* join ip & fraction*/
        div128 (a, scale, n, r);
        shiftdown (scale, 2);
        if (leq (scale, r, 2))
           add2 (n, one);
        shiftup (scale, 2);
        mul128 (n, scale, n);
        scale[0] = SCALE;
        scale[1] = 0;
        div128 (n, scale, a, b); /* split the 128-bit integer*/
        out->ip = a[0];     /* to integral part*/
        out->fp = (int) b[0]; /* fractional part*/     }
   if (neg)
      negate_real (out);}
///////////////////////////////////////////////////////////////////////////////
// ROUND$REAL ROUND A REAL TO A NUMBER OF FRACTIONAL DIGITS
///////////////////////////////////////////////////////////////////////////////
// Brief description: Round a real to a number of fractional digits
// Expected:
//           out      =    address of 64-bit real
//           in       =    address of 64-bit real
//           frac     =    number of fractional digits (scale to)
//   Result: void
// Example:
//  Round 176.357143 to 2 fractional digits = 176.36
//  round 176.357143 to -1 fractional digits = 180
//  scale 87.36 by 1 = 87.4
//  scale 87.36 by 0 = 87
//  scale 87.36 by -1 = 90
//  scale 87.36 by -2 = 100
//
///////////////////////////////////////////////////////////////////////////////
void routine round$real (real *out, real *_in, int frac)
{  real rin = *_in;
   real *in = &rin;
   int scale = SCALE;
   int rem;              /* remainder*/
   int neg = 0;
   if (is_negative_real (in))
     {        negate_real (in);
        neg = 1;      }
   if (frac < -SCALE_DIGITS)
      frac = -SCALE_DIGITS;
   if (frac > SCALE_DIGITS)    /* our maximum precision*/
      frac = SCALE_DIGITS;
   /*  *  positive 'frac' then only scale the 'fp' part*/
   if (frac == 0)
     {        out->ip = in->ip;     /* copy over ip part*/
        if ((SCALE >> 1) <= in->fp) /* 'fp' part greater than a half*/
           out->ip += 1;       /* carry one*/
        out->fp = 0;       /* no decimal digits*/     }
```

```
else if (frac > 0)
  {       out->ip = in->ip;     /* copy over ip part*/
    scale /= ten_power[frac];
    rem = in->fp % scale; /* get remainder*/
    out->fp = in->fp / scale;   /* scale down*/
    if ((scale >> 1) <= rem) /* is there a carry?*/
       out->fp += 1;      /* carry one*/
    out->fp *= scale;     /* and up again (to get rid of lower digits)*/
    if (out->fp >= SCALE)
     {              /* overflow ?*/
        out->fp -= SCALE;
        out->ip++;      /* carry one*/
     }     }
else
  {                     /* scale the 'ip' part*/
    int64 ip = in->ip;
    if ((SCALE >> 1) <= in->fp) /* 'fp' part greater than a half*/
       ip += 1;         /* carry one*/
    out->fp = 0;        /* zero result fp part*/
    scale = ten_power[-frac];   /* a ten_power*/
    rem = (int) (ip % scale);   /* get remainder*/
    out->ip = ip / scale; /* scale down*/
    if ((scale >> 1) <= rem) /* is there a carry?*/
       out->ip += 1;      /* carry one*/
    out->ip *= scale;     /* and up again (to get rid of lower digits)*/     }
 if (neg)
    negate_real (out);}
```

// SCALE$REAL SCALE A REAL TO A NUMBER OF FRACTIONAL DIGITS

```
// Brief description: Scale a real to a number of fractional digits
// Expected:
//          out     =   address of 64-bit real
//          in      =   address of 64-bit real
//          frac    =   number of fractional digits (scale to)
//   Result:  void
void scale$real (real * out, real * in, int frac)
{  round$real ((real *) out, (real *) in, -frac);}
char *dotptr;
char *cvtnumptr;
static int zeros, leader, frac, sci, period, neg, pad, leadzero, e;
```

// FL TO STRING FRAC

```
// Brief description: Convert a float to a string
// Expected: val    - value to be converted to a string
//          outptr - an address of conversion.
//          len    - maximum length of conversion
//          digits - the number of fractional digits
//          flags:
int fl_string_output (int *len, char *outptr);
int routine new_fl_string_output (int *len, char *outptr, int flags);
int routine fl_to_string_frac (int *len, char *outptr, real * val, int fdigits,
                               int flags)
{  char tmpbuf[cvtnumlen];
   DESC_S tmpdsc[1];
   real rvalue;
   int status;
   realinfo rinfo;
   int is_implied_dec = (flags & 2);
   int *s = (int*)&val->ip;
   *outptr = '\0';
   if (len[0] <= 1)
      return 0;
   set$desc (tmpdsc, cvtnumlen, tmpbuf);
   pad = (flags & cvt_pad) != 0;
   rvalue = *val;
   s = (int*)&val->ip;
   new_cnvout (&rvalue, tmpdsc, fdigits, &rinfo, flags, *len);
   if (tmpbuf[0] == '*')
      return 0;
   zeros = fdigits - rinfo.fractdigits;
```

```
   dotptr = &tmpbuf[cvtnumlen - rinfo.rlen]; // point us to the first data
   neg = rinfo.neg;
   if (neg)
      dotptr = &dotptr[1]; // skip sign  (??dme?? fix this later)
   sci = FALSE;
   leader = rinfo.wholedigits;
   period = leadzero = 0;
   frac = fdigits;            // rinfo.fractdigits;
   //?? fix  & 2 to be "IMPLIED_DECIMAL"
   if (is_implied_dec)
      status = new_fl_string_output (len, outptr, flags);
   else
      status = fl_string_output (len, outptr);
   if (!status)
      fill (len[0], outptr, '*');
   return 1;}                              // fl_to_string_frac
///////////////////////////////////////////////////////////////////////////
// CVT OUT D F
///////////////////////////////////////////////////////////////////////////
// Brief description: Convert real to floating ASCII output.
// Expected:
//             fval   =   address of 64-bit real
//             dlen   =   fill length
//             desc   =   pointer to output descriptor
//   Result:  status from fl_to_string_frac
///////////////////////////////////////////////////////////////////////////
int routine cvt_out_d_f (real * fval, int dlen, void *desc)
{  int status = SUCCESS;
   int len;
   char *text;
   char numbuf[100];
   split$desc (desc, &len, &text);
   round$real (fval, fval, 0);
   status = fl_to_string_frac (&len, numbuf, fval, 0, 0);
   if (!status)
      return MSG_CVTERROR;
   // len -= 1;
   fill (dlen, text, ' ');
   copy (len, numbuf, text + dlen - len);
   return status;}                             // cvt_out_d_f
///////////////////////////////////////////////////////////////////////////
// REAL CVT OUT D F
///////////////////////////////////////////////////////////////////////////
// Brief description: Convert real to floating ASCII output.
// Expected:
//             fval   =   address of 64-bit real
//             dlen   =   fill length
//             desc   =   pointer to output descriptor
//   Result:  status from fl_to_string_frac
///////////////////////////////////////////////////////////////////////////
int routine real_cvt_out_d_f (real * fval, int dlen, void *desc)
{  int status = SUCCESS;
   int len;
   char *text;
   char numbuf[100];
   char *p = numbuf;
   split$desc (desc, &len, &text);
   round$real (fval, fval, 0);
   status = fl_to_string_frac (&len, numbuf, fval, 0, 0);
   if (!status)
      return MSG_CVTERROR;
   fill (dlen, text, ' ');
   if (dlen < len)
     {        p += len - dlen;
        len = dlen;     }
   copy (len, p, text + dlen - len);
   return status;}
///////////////////////////////////////////////////////////////////////////
// NAT_CVT_STR_FLT CONVERT ASCII FLOATING STRING TO REAL
///////////////////////////////////////////////////////////////////////////
// Brief description: Convert ascii floating string to real
// Expected:
//             result =   address of result 64-bit real
//             dlen   =   length of ascii text
```

```
//            text    =   ascii input text
//   Result:  status
//////////////////////////////////////////////////////////////////////////////////
int routine math_cvt_str_flt (real * result, int len, char *text)
{  real n = 0;
   int neg = 0;
   int e = 0;
   int i;
   for (i = 0; i < len && text[i] && text[i] <= ' '; i++);  /* skip leading*/
   if (text[i] == '-')
     {         neg = 1;
       i++;        }
   for (; i < len && text[i] && text[i] <= ' '; i++); /* skip leading*/
   if (i >= len)
     {         *result = n;
       return 1;      }
   if (!(tolower (text[i]) >= '0' && tolower (text[i]) <= '9'))
     {                          // not numeric
       switch (tolower (text[i]))
         {           case 'e':
           case 'x':
           case '+':
           case '.':
             break;
           default:
             return 0;
         }      }
   /*  *  get the integral part first*/
   for (; i < len && text[i] >= '0' && text[i] <= '9'; i++)
     {         n.ip *= 10;
       n.ip += text[i] & 0x0f;      }
   if (i < len && tolower (text[i]) == 'e')
     {         e = 0;
       for (i++; i < len && text[i] >= '0' && text[i] <= '9'; i++)
         {           e *= 10;
           e += text[i] & 0x0f;         }
       while (e--)
         n.ip *= 10;      }
   if (i < len && text[i] == '.')
     {         int f = 0;
       int p = ++i;
       if (i < len && (text[i] < '0' || text[i] > '9'))
         return 0;
       for (;
            i < len && i < p + SCALE_DIGITS && text[i] >= '0'
            && text[i] <= '9'; i++)
         {            f *= 10;
            f += text[i] & 0x0f;         }
       /* do not round ascii input*/
       /* if (i == p+6 && text[i] >= '5' && text[i] <= '9')*/
       /*  f += 1;                                       */
       for (; i < p + SCALE_DIGITS; i++)
         f *= 10;
       n.fp = f;     }
   if (i < len && tolower (text[i]) == 'e')
     {         e = 0;
       for (i++; i < len && text[i] >= '0' && text[i] <= '9'; i++)
         {           e *= 10;
           e += text[i] & 0x0f;         }
       while (e--)
         {           n.fp *= 10;
           n.ip *= 10;
           if (n.fp >= SCALE)
             {               n.fp -= SCALE;
               n.ip++;
             }
         }     }
   if (is_negative_real (&n))
     {         runtime$error (MSG_FLOATOVF);  /* overflow*/      }
   if (neg)
     negate_real (&n);
   *result = n;
```

```
   return 1;}
int routine ator(real *a, char *b)
{  return math_cvt_str_flt(a, strlen(b), b);}
//////////////////////////////////////////////////////////////////////
// NAT_CVT_STR_FLT_IMPDEC CONVERT ASCII FLOATING STRING TO REAL
//////////////////////////////////////////////////////////////////////
// Brief description: Convert ascii floating string to real
//
// Expected:
//             result =   address of result 64-bit real
//             dlen   =   length of ascii text
//             text   =   ascii input text
//   Result:  status
//////////////////////////////////////////////////////////////////////
int routine math_cvt_str_flt_impdec(real * result, int len, char *text, int fdigits)
{  real n = 0;
   int neg = 0;
   int i;
   for (i = 0; i < len && text[i] && text[i] <= ' '; i++);   /* skip leading*/
   if (text[i] == '-')
     {        neg = 1;
        i++;        }
   for (; i < len && text[i] && text[i] <= ' '; i++); /* skip leading*/
   if (i >= len)
     {         *result = n;
        return 1;      }
   /*  *  get the integral part first*/
   for (; i < len-fdigits && text[i] >= '0' && text[i] <= '9'; i++)
     {         n.ip *= 10;
        n.ip += text[i] & 0x0f;       }
   int f = 0;
   int p = i;
   if (i < len && (text[i] < '0' || text[i] > '9'))
           return 0;
   for (; i < len && i < p + SCALE_DIGITS && text[i] >= '0'
            && text[i] <= '9'; i++)
          {          f *= 10;
            f += text[i] & 0x0f;         }
   for (; i < p + SCALE_DIGITS; i++)
      f *= 10;
    n.fp = f;
   if (is_negative_real (&n))
     {         runtime$error (MSG_FLOATOVF);  /* overflow*/      }
   if (neg)
      negate_real (&n);
   *result = n;
   return 1;}
//////////////////////////////////////////////////////////////////////
// DTOR CONVERT A DOUBLE TO REAL
//////////////////////////////////////////////////////////////////////
// Brief description: Convert a double to a 128-bit real
// Expected:
//             a    =   address of input double.
//             b    =   address of output 128-bit real
//   Result:  void
//////////////////////////////////////////////////////////////////////
void
routine dtor (double *a, real * b)
{    double d = *a;
    double my_d;
    int64 ip;
    int fp;
    if (d > LARGEST_NUMBER || d < -LARGEST_NUMBER)
           runtime$error (MSG_FLOATOVF);
    // handle rounding now by adding in 5*10^-8
    if (d < 0)
       my_d = d - 0.000000005;
    else
       my_d = d + 0.000000005;

    ip = (int64) my_d;
    my_d -= ip;
    fp = (int)(my_d * SCALE);  // this is the same code I last sent you
    fp = (fp/10)*10;       // truncates at SCALE-1 digits
```

```
    if (ip && fp < 0) /* if ip then*/
        fp = -fp;      /* fp always positive*/
    b->ip = ip;
    b->fp = fp;
    }
```

// MULR MULTIPLY TWO REALS

```
// Brief description: Multiply two 128-bit integers. This is used for intermediate
//   arithmetic.
// Expected:
//            _a  =   address of real (_a * _b)
//            _b  =   address of real (_a * _b)
//            r   =   address of real (result)
//   Result:  void
void mulr (real * _a, real * _b, real * _c)
{  r96multiply(_a, _b, _c);}
```

// CVT$QUAD$FLT CONVERT A 64-BIT QUADWORD TO REAL

```
// Brief description: Convert a 64-bit quadword to a real.
// Expected:
//            result   =   address of real
//            p_quad   =   address of 64-bit quadword
//   Result:  status
int routine cvt$quad$flt (real * result, int *p_quad)
{  result->ip = (*(int64 *) p_quad);
   result->fp = 0;    // Clear fractional portion ++dme++
   return 1;}
```

// FL TO STRING

```
// Brief description: Convert a float to a string
// Expected: val    - value to be converted to a string
//           outptr - an address of conversion.
//           len    - maximum length of conversion
//           digits - the number of significant digits
//           flags:
int routine fl_to_string (int *len, char *outptr, real * val, int digits,
                   int flags)
{  char tmpbuf[cvtnumlen];
   DESC_S tmpdsc[1];
   int status;
   realinfo rinfo;
   pad = (flags & cvt_pad) != 0;
   set$desc (tmpdsc, cvtnumlen, tmpbuf);
   new_cnvout (val, tmpdsc, digits, &rinfo, 0, *len);
   if (tmpbuf[0] == '*')
      return 0;
   zeros = digits - rinfo.fractdigits;
   dotptr = &tmpbuf[cvtnumlen - rinfo.rlen]; // point us to the first data
   neg = rinfo.neg;
   if (neg)
      dotptr = &dotptr[1]; // skip sign  (??dme?? fix this later)
   sci = FALSE;
   leader = rinfo.wholedigits;
   period = leadzero = 0;
   frac = rinfo.fractdigits;
   cvtnumptr = tmpbuf;
   status = fl_string_output (len, outptr);
   return status;}                       // fl_to_string
```

// FL STRING OUTPUT

```
// Brief description: Convert a float to a string
//  The result looks like:    [- ]nnnnnnn.ffffff
//  Where [- ] is either a space or '-'
//  nnnn = whole number, ffff=fraction
//  (notice how the decimal point is included)
// Expected: val    - value to be converted to a string
//           outptr - an address of conversion.
//           len    - maximum length of conversion
```

```
//          digits - the number of significant digits
//          flags:
/////////////////////////////////////////////////////////////////////////////////////////////
int routine fl_string_output (int *len, char *outptr)
{#define $save_lit(ch) \
    {  \
    if ((outptr - outbegin) > maxlen)  \
        return 0;                                  \
    outptr[0] = ch; \
    outptr++;          \
    }
#undef $save
#define $save(tlen, txt) \
    {           \
    if ((outptr - outbegin) > maxlen)  \
        return 0;                                  \
    copy(tlen, txt, outptr);                     \
    outptr += tlen;                                   \
    }
   char *outbegin;
   int maxlen;
   memset(outptr, 0, *len);      /* null fill result*/
   maxlen = len[0];
   outbegin = outptr;
   if (neg)
      $save_lit ('-')
      else
   if (pad)
      $save_lit (' ');
   if (leadzero)
      $save_lit ('0');
   $save (leader, &dotptr[0]);
   if (frac == 0 && leader == 0 && !leadzero)
      $save_lit ('0');
   if (frac > 0)
     {        $save_lit ('.');
        $save (frac, &dotptr[1 + leader]);      }
   if (pad)
      $save_lit (' ');
   *len = outptr - outbegin;
   if (maxlen < *len)
      return 0;
   return 1;}                           // fl_string_output
/////////////////////////////////////////////////////////////////////////////////////////////
// FL STRING OUTPUT
/////////////////////////////////////////////////////////////////////////////////////////////
// Brief description: Convert a float to a string
//  The result looks like:   [- ]nnnnnnnffffff
//  Where [- ] is either a space or '-'
//  nnnn = whole number, ffff=fraction
//  (notice how the decimal point is implied but not included)
// Expected: val    - value to be converted to a string
//           outptr - an address of conversion.
//           len    - maximum length of conversion
//           digits - the number of significant digits
//           flags:  2 = implied decimal digits
/////////////////////////////////////////////////////////////////////////////////////////////
int routine new_fl_string_output (int *len, char *outptr, int flags)
{  char *outbegin;
   int maxlen;
   int is_implied_dec = FALSE;
   if (flags & 2)
      is_implied_dec = TRUE;
   maxlen = len[0];
   outbegin = outptr;
   if (neg)
      $save_lit ('-')
      else
   if (pad)
      $save_lit (' ');
   if (leadzero)
      $save_lit ('0');
   $save (leader, &dotptr[0]);
```

```
   if (frac == 0 && leader == 0 && !leadzero)
      $save_lit ('0');
   if (frac > 0)
     {        if (is_implied_dec)    //??dme?? as WHy no ";" on line below??
           $save (frac, &dotptr[leader])  // just the decimal digits...no "."
              else
           $save (frac + 1, &dotptr[leader]);    // fract+1 to include the "."     }
   if (pad)
      $save_lit (' ');
   *len = outptr - outbegin;
   if (maxlen < *len + is_implied_dec)
      return 0;
   return 1;}                        //  new_fl_string_output
/////////////////////////////////////////////////////////////////////////////////////
// FL TO STRING
/////////////////////////////////////////////////////////////////////////////////////
// Brief description: Convert a float to a string
// Expected: val    - value to be converted to a string
//          outptr - an address of conversion.
//          len    - maximum length of conversion
//          digits - the number of significant digits
//          flags:
/////////////////////////////////////////////////////////////////////////////////////
int routine real_fl_to_string (int *len, char *outptr, real * val, int digits,
                        int flags)
{  char tmpbuf[cvtnumlen];
   DESC_S tmpdsc[1];
   int d = 0;
   int i;
   pad = (flags & cvt_pad) != 0;
   set$desc (tmpdsc, cvtnumlen, tmpbuf);
   math_cnvout (val, tmpdsc, digits);
   if (pad)
     {        *outptr++ = ' ';
         d = 1;     }
   for (i = 0; tmpbuf[i]; i++, d++)
      *outptr++ = tmpbuf[i];
   if (pad)
     {        d++;
         *outptr++ = ' ';       }
   *len = d;
   e = 0;
   frac = 0;                 /* leader - zeros;*/
   sci = 0;
   leader = 0;
   period = leadzero = 0;
   return 1;}                      // fl_to_string
/////////////////////////////////////////////////////////////////////////////////////
// FL TO STRING FRAC
/////////////////////////////////////////////////////////////////////////////////////
// Brief description: Convert a float to a string
// Expected: val    - value to be converted to a string
//          outptr - an address of conversion.
//          len    - maximum length of conversion
//          digits - the number of fractional digits
//          flags:
//  if (!fl_to_string_frac (&numlen, buffer, (int*)&number,
//                             form->form$fraclen, 0))
//      return FALSE;
/////////////////////////////////////////////////////////////////////////////////////
int routine real_fl_to_string_frac (int *len, char *outptr, real * val,
                          int digits, int flags)
{  char tmpbuf[cvtnumlen];
   DESC_S tmpdsc[1];
   char *e = outptr + *len;
   char *p = outptr;
   char fill = '0';          /* default*/
   int i, n;
   if (len[0] <= 1)
      return 0;
   pad = (flags & cvt_pad) != 0;
   set$desc (tmpdsc, sizeof (tmpbuf), tmpbuf);
   math_cnvout (val, tmpdsc, digits);
```

```
    if (pad)
        *p++ = ' ';
    for (i = 0; tmpbuf[i] && p < e && tmpbuf[i] != '.'; i++)
        *p++ = tmpbuf[i];
    if (digits)
      {         if (tmpbuf[i] != '.')
            *p++ = '.';
        else
            *p++ = tmpbuf[i++];
        for (n = 0; n < digits && tmpbuf[i] && p < e; i++, n++)
            *p++ = tmpbuf[i];
        for (; n < digits && p < e; n++)
            *p++ = '0';
        fill = ' ';      }
    else
      {         if (tmpbuf[i] != '.')
          {              --e;
            fill = ' ';       }
        for (; tmpbuf[i] && p < e; i++)
            *p++ = tmpbuf[i];      }
    if (pad)
        *p = ' ';
    if (p < e)
      {                      /* move to end of string*/
        while (p > outptr)
            *--e = *--p;
        while (e > outptr)
            *--e = fill;     }
    return 1;}                         // fl_to_string_frac
//////////////////////////////////////////////////////////////////////////////////
// FL STRING OUTPUT
//////////////////////////////////////////////////////////////////////////////////
// Brief description: Convert a float to a string
// Expected: val    - value to be converted to a string
//           outptr - an address of conversion.
//           len    - maximum length of conversion
//           digits - the number of significant digits
//           flags:
//////////////////////////////////////////////////////////////////////////////////
int routine real_fl_string_output (int *len, char *outptr)
{   char *outbegin;
    int maxlen;
    maxlen = len[0];
    outbegin = outptr;
    if (neg)
        $save_lit ('-')
        else
    if (pad)
        $save_lit (' ');
    if (leadzero)
        $save_lit ('0');
    $save (leader, &dotptr[1]);
    if (frac == 0 && leader == 0 && !leadzero)
        $save_lit ('0')
        else
    if (frac > 0)
      {         $save_lit ('.');
        period = 0;
        if (!sci)
            while (e < 0)
             {              e++;
             $save_lit ('0')}
        $save (frac, &dotptr[1 + leader]);       }
    else if (!sci)
      {         while (e - leader > 1)
          {              e--;
          $save_lit ('0')}       }
    if (period)
        $save_lit ('.');
    if (sci)
        $save (4, &cvtnumptr[cvtnumlen - 4]);
    if (pad)
```

```
      $save_lit (' ');
   *len = outptr - outbegin;
   return 1;}                      // fl_string_output
#include "tmath.h"
////////////////////////////////////////////////////////////////////////
// NAT INT SIN
////////////////////////////////////////////////////////////////////////
// Brief description: Returns the SINE of an angle specified in radians
// Expected:  in  - passed real an angle
//   Result:  out - sine of given angle
////////////////////////////////////////////////////////////////////////
double mth$dsin (double *);
c_routine int math_sin (double *in, double *out)
{  double a;
   real b;
   a = getdouble ((real *) in);
   b = mth$dsin (&a);
   *(real *) out = b;
   return 1;}                      // math_sin
////////////////////////////////////////////////////////////////////////
// NAT INT COS
////////////////////////////////////////////////////////////////////////
// Brief description: Returns a cosine  of an angle.
// Expected:  in    - passed real angle
//   Result:  out   - cosine ofd an angle.
////////////////////////////////////////////////////////////////////////
double mth$dcos (double *);
c_routine int math_cos (double *in, double *out)
{  double a;
   real b;
   a = getdouble ((real *) in);
   b = mth$dcos (&a);
   *(real *) out = b;
   return 1;}                      // math_cos
////////////////////////////////////////////////////////////////////////
// NAT INT TAN
////////////////////////////////////////////////////////////////////////
// Brief description: Returns a tanget of an angle.
// Expected:   in   - passed real angle
//   Result:   out  - tanget of an angle
////////////////////////////////////////////////////////////////////////
double mth$dtan (double *);
c_routine void math_tan (real * in, real * out)
{  double a, b;
   real r;
   a = getdouble (in);
   b = mth$dtan (&a);
   dtor (&b, &r);
   *out = r;}                      // math_tan
////////////////////////////////////////////////////////////////////////
// NAT INT ASIN
////////////////////////////////////////////////////////////////////////
// Brief description: ASIN(x) returns the angle whose SIN is x.
// Expected:  in    - passed real value
//   Result:  out   - angle
////////////////////////////////////////////////////////////////////////
double mth$dasin (double *);
c_routine void math_asin (double *in, double *out)
{  double a;
   real b;
   a = getdouble ((real *) in);
   b = mth$dasin (&a);
   *(real *) out = b;}                      // math_asin
////////////////////////////////////////////////////////////////////////
// NAT INT ACOS
////////////////////////////////////////////////////////////////////////
// Brief description: ACOS(x) returns the angle whose COS is x
// Expected:  in   - passed real value
//   Result:  out  - angle .
////////////////////////////////////////////////////////////////////////
double mth$dacos (double *);
c_routine void math_acos (double *in, double *out)
{  double a;
   real b;
   a = getdouble ((real *) in);
   b = mth$dacos (&a);
   *(real *) out = b;}                      // math_acos
////////////////////////////////////////////////////////////////////////
```

```
// NAT INT ATAN
////////////////////////////////////////////////////////////////////////////////
// Brief description: ATAN(x) returns the angle whose TANGENT is x.
// Expected:    in  - passed real value.
//   Result:    out - angle.
////////////////////////////////////////////////////////////////////////////////
double mth$datan (double *);
c_routine void math_atan (double *in, double *out)
{  double a;
   real b;
   a = getdouble ((real *) in);
   b = mth$datan (&a);
   *(real *) out = b;}                        // math_atan
////////////////////////////////////////////////////////////////////////////////
// NAT INT SINH
////////////////////////////////////////////////////////////////////////////////
// Brief description: SINH(X) returns hyperbolic sine of X.
// Expected:   in   -  passed real value.
//   Result:   out  -  hyperbolic sine of passed value
////////////////////////////////////////////////////////////////////////////////
double mth$dsinh (double *);
c_routine void math_sinh (double *in, double *out)
{  double a;
   real b;
   a = getdouble ((real *) in);
   b = mth$dsinh (&a);
   *(real *) out = b;}                        // math_sinh
////////////////////////////////////////////////////////////////////////////////
// NAT INT COSH
////////////////////////////////////////////////////////////////////////////////
//  Brief dexcription:
//    COSH(X) returns hyperbolic cosine of X.
// Expected:   in    - passed real value.
//   Result:   out   - hyperbolic cosine of a passed value.
////////////////////////////////////////////////////////////////////////////////
double mth$dcosh (double *);
c_routine void math_cosh (double *in, double *out)
{  double a;
   real b;
   a = getdouble ((real *) in);
   b = mth$dcosh (&a);
   *(real *) out = b;}                        // math_cosh
////////////////////////////////////////////////////////////////////////////////
// NAT INT TANH
////////////////////////////////////////////////////////////////////////////////
// Brief description: TANH(X)  returns hyperbolic tangent of X.
// Expected:   in   - passed real value.
//   Result:   out  - hyperbolic tanget of passed value.
////////////////////////////////////////////////////////////////////////////////
double mth$dtanh (double *);
c_routine void math_tanh (double *in, double *out)
{  double a;
   real b;
   a = getdouble ((real *) in);
   b = mth$dtanh (&a);
   *(real *) out = b;}                        // math_tanh
////////////////////////////////////////////////////////////////////////////////
// NAT RND
////////////////////////////////////////////////////////////////////////////////
// Brief description: // Expected:  value - numeric expression  seed = global seed value
//   Result: Returns a random  number between one and passed //          numeric expression.
////////////////////////////////////////////////////////////////////////////////
void realip (real *, real *);
real mth$random (int *);
void math_rnd (real * value)
{  real f = mth$random (&seed);
   *value = f;}                         // math_rnd
////////////////////////////////////////////////////////////////////////////////
// NAT SGN
////////////////////////////////////////////////////////////////////////////////
// Brief description: Returns the sign of a number
// Expected:  in  - passed numeric expression
//   Result:  out - sign of a a numeric expression
//               = +1 if expression > 0
//               = -1 if expression < 0
//               =  0 if expression = 0
////////////////////////////////////////////////////////////////////////////////
c_routine int mth$sgn (int *);
```

```
c_routine int realsgn (void *);
void math_sgn (real * in, int *out)
{  real n = 0;
   out[0] = $cmp$real (in, &n);}                        // math_sgn
////////////////////////////////////////////////////////////////////////
// NAT INT ABS
////////////////////////////////////////////////////////////////////////
// Brief expression:
//  Returns the absolute value of a numeric expression.
// Expected: in  - passed real expression
//   Result: out - the absolute value of real expression
////////////////////////////////////////////////////////////////////////
double mth$dabs (double *);
void math_abs (real * in, real * out)
{  double a;
   real b;
   a = getdouble ((real *) in);
   b = mth$dabs (&a);
   *(real *) out = b;}                          // math_abs
////////////////////////////////////////////////////////////////////////
// NAT REAL MAX
////////////////////////////////////////////////////////////////////////
// Brief description: Returnes the larger of two reals X and Y.
// Expected:  x  - real value
//            y  - real value
//   Result:  out - larger of two reals X and Y
////////////////////////////////////////////////////////////////////////
void math_real_max (real * x, real * y, real * out)
{  if ($cmp$real (x, y) < 0)
      *out = *y;
   else
      *out = *x;}                          // math_real_max
////////////////////////////////////////////////////////////////////////
// NAT REAL MIN
////////////////////////////////////////////////////////////////////////
// Brief description: Returns the lesser of the two reals X and Y
// Expected: x   - passed real value //          y   - passed real value //   Result: out - lesser of two reals X and Y
////////////////////////////////////////////////////////////////////////
void math_real_min (real * x, real * y, real * out)
{  if ($cmp$real (x, y) < 0)
      *out = *x;
   else
      *out = *y;}                          // math_real_min
////////////////////////////////////////////////////////////////////////
// NAT SQR
////////////////////////////////////////////////////////////////////////
// Brief description: Returns a square root of a real value  // Expected:  in  - passed real number
//   Result:  out - square root
////////////////////////////////////////////////////////////////////////
double mth$dsqrt (double *);
void math_sqr (real * in, real * out)
{  double a;
   real b;
   a = getdouble (in);
   b = mth$dsqrt (&a);
   *out = b;}                          // math_sqr
////////////////////////////////////////////////////////////////////////
// NAT INT LOG
////////////////////////////////////////////////////////////////////////
// Brief description: Returns the natural logarithm of a real number // Expected:   in - passed real value
//   Result:   out- natural logarithm  //
////////////////////////////////////////////////////////////////////////
double mth$dlog (double *);
void math_log (real * in, real * out)
{  double a;
   real b;
   a = getdouble (in);
   b = mth$dlog (&a);
   *out = b;}                          // math_log
////////////////////////////////////////////////////////////////////////
// NAT LOG2
////////////////////////////////////////////////////////////////////////
// Brief description: Returns  the base 2 logarithm of given real value
// Expected:  in  -  passed real value  //   Result:  out -  base 2 logarithm
////////////////////////////////////////////////////////////////////////
double mth$dlog2 (double *);
void math_log2 (real * in, real * out)
```

```
{  double a;
   real b;
   a = getdouble (in);
   b = mth$dlog2 (&a);
   *out = b;}                        // math_log2
//////////////////////////////////////////////////////////////////////////////////////////////////
// NAT INT LOG10
//////////////////////////////////////////////////////////////////////////////////////////////////
// Brief description: Returns the common logarithm of the real value
// Expected:  in  - passed real value  //   Result:  out - common logarithm
//////////////////////////////////////////////////////////////////////////////////////////////////
double mth$dlog10 (double *);
void math_log10 (real * in, real * out)
{  double a;
   real b;
   a = getdouble (in);
   b = mth$dlog10 (&a);
   *out = b;}                        // math_log10
```